Object Persistence

Stefan Chyssler scr02001@student.mdh.se   Johan Krüger jk02002@student.mdh.se

22nd May 2006
Abstract

Almost every application needs a way to make its data persistent. Even the simplest application usually has at least some settings that need to be stored. There are a lot of ways to achieve this and their suitability depends on the amount of data to be stored, the complexity of that data and performance requirements. This report is meant to be an overview of different techniques to achieve the much needed object persistence in an application. The report is written from an application developer perspective. The underlying application is assumed to be a web shop and therefore does not discuss aspects of object persistence present in embedded systems. The report will also introduce the reader to the Data Access Object design pattern. This design pattern makes it easier to migrate an application between different persistence implementations. The object persistence techniques within the report touch varied technologies such as binary files, XML and XQuery, relational database and SQL, object-relational mapping tools and object oriented databases. Using a relational database together with a potent object-relational is well suited for developing high concurrency systems such as the web-shop application. The different technologies are presented and discussed at a high level arguing strengths and weaknesses. The bottom line of this report is that the choice of persistence technique should always be based on the needs and requirements of the application.
## Contents

1 Introduction ........................................ 3

2 Object Persistence ................................... 4
   2.1 Binary files and XML ............................. 5
      2.1.1 Binary Files .................................. 5
      2.1.2 eXtensible Markup Language XML ............... 6
      2.1.3 XQuery Language ............................... 6
   2.2 Relational Databases and SQL ...................... 8
   2.3 Object Relational Mapping Tools ................. 11
      2.3.1 Mapping Classes to Tables with Configuration .. 11
      2.3.2 Convention over Configuration: Ruby on Rails .. 13
   2.4 Object Databases ................................ 15
      2.4.1 Native Queries ................................ 16

3 Design Pattern: Data Access Objects ............... 17

4 Discussion ........................................... 18
1 Introduction

Almost every application needs a way to make its data persistent. Even the simplest application usually has at least some settings that need to be stored. There are a lot of ways to achieve this and their suitability depends on the amount of data to be stored, the complexity of that data and performance requirements. For small, single user, desktop applications a plain binary file or an XML file may be enough. They are easy to create and use, sufficiently fast on small amounts of data, but not very reliable and definitely not well suited for multi-user applications. Also, it’s quite difficult to reuse the data or the code which saves and loads the data since they are usually designed for a specific application.

Database Management Systems (DBMS) are created with the intention to store and retrieve data as safe and efficient as possible, allowing multiple users. The most common type of DBMS as of today is the relational DBMS. These systems store data in simple two-dimensional tables which can be connected to each other using relations and identifiers. This is a very powerful mechanism that allows the user to retrieve single- or groups of items by issuing queries to the DBMS using a query language such as Structured Query Language (SQL). The problem with relational DBMS is that they don’t map very well to the popular object oriented programming concept. To store objects a specialized query has to be written for each type of object that needs to be stored in the application, which can be quite many. This is a repetitive, time-consuming and error prone task for a programmer. That is why other persistence techniques such as Object-Relational mapping tools and Object-Oriented Database Management Systems (OODBMS) have been developed. Object-Relational mapping tools can eliminate some or all of the coding needed to store objects by using automatic code generation.

Object-Oriented Database Management Systems (OODBMS) has been developed to overcome the mapping problem between object-oriented programming languages and the DBMS by allowing entire objects to be stored. These systems can store complete objects including other objects it has references to with only a single line of code. Data can be accessed and modified using methods of the class the object is constructed from, this way string queries are not needed. The advantage of using references (pointers) over queries is that they are usually faster at accessing objects. On the other hand it may be cumbersome and inefficient to retrieve larger collections of objects.

Ideally the objects should be stored and retrieved without any additional programming. How can this be done? Is it even possible? This report presents a number of different persistence technique as well as a design pattern that separate the persistence layer from the application layer. This separation can help when a project must change its persistence implementation.
2 Object Persistence

Using binary files was one of the first ways of storing persistent application data. But as program complexity increased, the requirement for better and more general persistence techniques grew. That’s why relational databases management systems were invented; they allowed huge amounts of data to be stored in tables using a standardized query language. It made it possible to easily reuse data or switch the underlying database system. When the object-oriented programming concepts became popular, it created a mismatch between the programming languages and the persistence layer. To overlap this gap, a couple of new persistence techniques have been developed: Object relational mapping tools which can automatically generate parts or all of the code needed to store objects. Other persistence techniques such as object databases that can store entire objects using the native programming language.

This report will introduce the reader to the following techniques for object persistence:

- Binary files and XML - Storing objects to the file system.
- Relational Databases and SQL - One of the most common techniques for data persistence.
- Object Relational Mapping Tools - Generative programming for relational databases.
- Object Databases - Storing objects as they are.

Throughout this report, examples will be presented that are based on an extremely simplified web shop application. The UML diagram below shows the classes used in the examples and how they are related. There are four types in the application: Address, User, Item, and Order. A user can place several orders and each order is associated with a user. The order may contain one or several items. This system is obviously not feature-complete and has several limitations, but it will be sufficient for the examples given in this report.
2.1 Binary files and XML

Using binary-, text- or XML files are common techniques to make small amounts of data persistent in an application. Each of these techniques has their advantages and disadvantages. In general the advantages are that they are easy to use and implement, they can store recursive data such as tree structures, data is usually loaded into the memory during runtime which makes it fast to operate on but memory consuming. However, security is bad and there is no native support for synchronization or transactions which makes them unsuitable for security-critical and multi-user applications. The techniques are serial in nature and therefore slow when saving/loading data; in the worst case all data must be processed. Because of these properties binary- and XML files are well suited for single-user applications that deal with reasonably small amounts of data, where data integrity and security is not important.

2.1.1 Binary Files

Storing data in binary files is often a special-purpose solution that is very space efficient with little or no data overhead. From the view of an application a binary file is expected to conform to a specific format where certain data types are stored in a certain order. To reuse the data stored in a binary file in a new application it must know exactly how the data is structured. This can be very difficult or even impossible if not the format specification or any source code is available. Problems also arise when an additional value must be added to the objects stored in the file. That forces the entire file to be converted, and worse, it breaks compatibility with older versions of the application.
2.1.2 eXtensible Markup Language XML

The extensible markup language (XML) is today a well known and widely used language for describing data. One of its primary purposes is to allow sharing of data between different systems. The language is standardized and maintained by the World Wide Web Consortium (W3C). By having a strict syntax and supporting only a few data types the extensible markup language can be easily parsed [Wik06c]. Nowadays most of the popular programming languages have built-in support for reading and writing XML-files. In an XML file the data is stored as plain text which means that it with little or no trouble can be modified manually, but it also makes it quite space demanding due to lack of compression or smart storage algorithms. Xml-files can be self-documenting since the data is stored between descriptive elements which are organized in an ordered tree structure. But some of the information used to describe the data is redundant and thereby causes overhead which is proportional to the number of objects stored in the XML-file. The conclusion is: XML-files are not suitable for storing large amounts of data.

2.1.3 XQuery Language

The XQuery language is a functional query language currently under development by W3C. The language is intended to provide the means of accessing data in XML-files much like SQL does for a relational database. Using XQueries data can be manipulated in any source that can be viewed as XML, for example office documents and relational databases [Wik06d]. In XQuery, data is located using path expressions the same way files are located in a file-system. Specific elements can be found using conditions or index numbers. The language also facilitates XML constructors and FLWOR (for, let, where, order by, return) expressions. XML constructors can be used to create new XML structures and FLWOR expressions represent the SELECT statement in SQL [Rob].

A table called 'Orders' in a relational database may be accessed using the following path expression to retrieve all the orders for the userid 'Johan'. Note that this code does not show how the database is opened in the XQuery language.

\[
\text{collection('Orders')/Order[UserId='Johan']}
\]

This expression may be translated to an SQL query similar to:

\[
\text{SELECT * FROM Orders WHERE UserId='Johan'}
\]

The FLWOR expression power in XQuery is demonstrated in the following example, where all items in all orders, with price greater than 1000, and placed by the UserId 'Johan' is returned. The result is sorted by price.

\[
\text{for $item in doc("Orders.XML")/Order[UserId='Johan']/item where $item/@price > 1000 order by $item/price return $item}
\]
In the above example the $item variable is assigned to each item in the resulting set of all items in all orders placed by 'Johan'. The where-condition is checked for each of these items and all items that satisfies the condition are returned. By comparing this query with its SQL equal it can be concluded that they are quite similar but the former may seem more readable.

```
SELECT Name, Price FROM
(OUTER JOIN Orders,Items ON Orders.OrderId = Items.OrderId)
WHERE UserId = 'Johan' AND Price > 1000 ORDER BY Price
```

The XQuery language is a general query language like SQL, and like SQL it is also string based. This implies that queries are flexible and can easily be generated by an application during runtime. But there is a price for the flexibility and that is the lack of syntax- and type checks from the compiler of the programming language. Since all database operations in these query languages are invoked using strings the compiler can not determine if the syntax is correct or not. Errors in XQueries are not detected until they are executed, making it error prone and more difficult to debug.

The advantage of using the XQuery language is that it provides a uniform interface to XML-files, relational databases and any source that can be viewed as XML. This allows different persistence formats to be treated the same way, an XML-based persistence can easily be exchanged for a relational database or they can be used in combination. The latter solution should be interesting for most web applications and distributed systems because they usually store their data in databases and communicate with their clients using XML. If XQueries were used in these systems they wouldn’t have to use two different languages to retrieve objects from the database and send them over to the client.
2.2 Relational Databases and SQL

Relational databases are perhaps what most people come to think of when they hear about a database. This may be because relational databases are one of the most common ways of storing and retrieving data, in any type of system. Relational databases are however not directly object oriented in nature, the relational model assumes that all data and associations between data can be represented as mathematical relations. The data stored is then operated upon by means of what is called relational calculus or algebra [EN00]. In a practical view or from an object oriented programmers view, relational databases use tables and columns to represent object-persistent data. Relations between tables are represented with keys that refer to keys of another table, this will be covered to some extent later in this report.

Designing tables for a database just by examining object definitions can be a complex task. Think of the User defined in the UML in previos section, the user also has an adress which is represented by a separate class with properties such as city, zip code and street adress. In this scenario the design of tables will give you a number of choices, either to have a new table that represents the adress and an association between the user table and the adress table. A second approach could be to store the adress in the same table as the user. There are usually more than one way to create tables to store your designer objects. However basic design rules exists, a relational database should strive to conform to the rules of normalization [EN00]. While on the subject of SQL; it should also be said that SQL does not always map very well to objects and class definitions. Recursive data structures such as trees are a common problem when using relational databases. While it is entirely possible to store a tree structure within a table using an extra field to denote the parent of a tuple. Storing and retrieving tree structures via SQL can be extremely cumbersome. One solution could be to first fetch the parent, then its children, then their children and so forth until the leaf nodes are reached. This solution creates large amount of communication between the application and the database.
The above database schema defines three tables within a database, the Items table have intentionally been omitted. These tables map quite well to the UML of the web-shop described earlier. The foreign key references can be viewed as associations between the different UML entities. UML is in fact often used to model databases and there are tools to generate schemas from UML diagrams [Wiki06b]. It is generally a good idea to use a database generated Id number as the primary key for each row in a table [EN00]. These id numbers should not have business semantics in an application and only be used to reference between entities in the database storage. The technique to generate and retrieve these unique id numbers varies between database management systems. The next code example shows how to use the SQL language to retrieve all orders related to a User.

```java
Query selectQuery = new Query("SELECT Id, Date FROM Orders WHERE Purchaser=" + user.getId());
Result tuples = selectQuery.executeQuery();
List<Order> allOrders = new List<Order>();
while(tuples.hasNext()) {
    Tuple tuple = tuples.next();
    Order order = new Order();
    order.setId(tuple.getInt(0));
    order.setPurchaser(user);
    allOrders.add(order);
}
for(Order order : allOrders) {
    Query itemQuantity = new Query("SELECT ItemId, COUNT(*) FROM OrderItems WHERE OrderId = " + order.getId() + "GROUP BY ItemId");
    Result itemQtyTuples = itemQuantity.executeQuery();
    while(itemQtyTuples.hasNext()) { 
        Tuple tuple = itemQtyTuples.next();
        /* Fetch Items from items-table, create correct Quantity, add to the order */
    }
}
Although the example above is pretty simple, not using powerful SQL constructs such as joins, it shows some of the repetitive tasks that developers face when using RDBMS and SQL. Mapping the the tuple information returned by the RDBMS back to a newly created object is both cumbersome and a potential source of errors. These errors include mistakes such as SQL syntax errors or not using the correct order for the fields. The example also shows a problem with the many-to-many relations. An Order is associated with a number of items, and any item may be part of many Orders. To facilitate this there must be a table in the middle describing how they relate to each other, introducing yet another query or additional complexity to the initial query. In addition to these problems, a developer must consider duplication issues. If an Item object with Id 1 is created and referenced by one Order, another Order is created that also references the Item with Id 1. Should they refer to the same object instance, or just the same row, thus having two Item objects. If they should refer to the same object, there must be checks to see if they have been previously loaded and these checks must be called every time a new object is to be loaded, leaving the application with another source of potential errors.

While the SQL language itself is not necessarily bad, it is the mapping back and forth between objects in the programming language that most programmers seem to feel is most painstaking. In an interview Gavin King, founder of the Hibernate project (which is discussed later), said that if you spend some time thinking about the relational model and relational modeling, you will realize that it is very elegant. What makes it dirty is the mismatch problem, the mapping problem and that is the only part that makes it dirty. Not writing SQL, not working with SQL [Kin06].
2.3 Object Relational Mapping Tools

For many object-oriented applications there is really no need to use the full expressive powers of SQL, at least not all the time. An application may only need to be able to store and retrieve objects via Id numbers, or for Users the username (which could be the unique Id of the User type). Also performance nowadays is not just measured in execution time but rather how fast an application can be built, with less faults. A solution to a problem you had three of months ago is perhaps not that good it justifies the delay. It is therefore no surprise that there are many different tools that help developers reduce the amount of manual coding needed to persist their data.

The goal of these tools are usually to achieve some kind of transparent persistency, with varying levels of success. Tools may also be able to care of the differences between SQL dialects for different database vendors. What transparent persistency means and how it is achieved may differ between the different tools and framework. As an example the Hibernate O/RM framework [Hib06b] avaible for Java and .NET uses configuration files to map class definitions to tables. These configuration files are read at application startup and the framework can use byte-code instrumentation to modify the class definition at runtime.

2.3.1 Mapping Classes to Tables with Configuration

```xml
<hibernate-mapping>
  <class name="User" table="User">
    <id name="id" type="integer">
      <generator class="native"/>
    </id>
    <property name="name" type="String" column="name" />
    <!-- All orders that are associated with this user -->
    <set name="orders" lazy="true">
      <key>
        <column name="Purchaser" not-null="true" />
      </key>
      <one-to-many class="Order" />
    </set>
  </class>
</hibernate-mapping>
```

The configuration file above shows how to map the User class to a User table defined within a relational database. The configuration file should be pretty self explanatory, the syntax uses XML tags which denotes what class name maps to what table, which is the identity property and other properties. The most interesting is perhaps the set tag, in this case this denotes a one-to-many relation with the class Orders. It also specifies that this property should be lazy evaluated. Lazy evaluation means that the class will be modified at runtime so that when a call to the get-method for an object, the database is accessed, creating the set on the fly. By default Hibernate use JavaBeans-style property accessors(getters and setters), thus the `<property name="name" ...>` makes Hibernate use getName and setName methods. It should be noted though that Hibernate does not require public visibility and therefore does not break encapsulation. The magic behind lazy evaluation of the getOrders code is transparent to the developer, it cannot be seen when looking at the definition of the User class. What a developer would see in the User class method is just a plain old method, as shown below.
In an application using Hibernate there will be mapping files for all other persistence types. When all configuration files have been created it is time to use them to retrieve and persist your objects. The code below shows an example of retrieving all the orders and the items associated with these orders as well as showing some of the benefits of using a tool such as Hibernate to handle the persistence of objects. It is however important to note that this code does not show how to open up a Hibernate session and retrieve the User instance.

```java
setOrders(Set<Order> orders) {
    this.orders = orders;
}

public Set<Order> getOrders() {
    return this.orders;
}
```

It should be clear for any reader that the code in the example above is much more readable and elegant than the code, with almost the same semantics, in the Relational Database section. The lazy loading of orders specified in the User mapping file ensures that we do not load all the Order objects until we really need them. Similarly, the OrderItems associated with an Order may be lazy loaded. While lazy loading is a powerful technique, it should be used with care. If the session that were used to create the object has been closed the object will have no way of loading the property from the database. Leaving the session open however breaks the rule of thumb of opening the session and then closing it as soon as you are done with it [WACBL03].

Tools such as hibernate can increase the productivity of a developer as well as decrease possible errors. Persistent objects can be treated as any normal object and the developer does not have to re-invent and re-implement features such as lazy loading. While this is entirely possible for a developer to do, using free tools makes it an unnecessary task. Another feature common to generative tools is that if all tables have been set up correctly, using foreign keys whenever an association is present, these tools can be used to automatically create both configuration files and the source code used to represent the objects. Generative programming such as this further ensure that less faults in the persistency layer has been introduced into the application.

Hibernate is by far the only potent O/RM tool, there exists a lot of similar tools each design with a different kind of philosophy. An interested reader should search the Internet for mapping tools suitable for his or her favourite programming language. While it is possible to configure yourself to death with the Hibernate framework, there are tools that take a different approach. One such is the Active Records within the Ruby on Rails framework, which will be discussed in the next section.
2.3.2 Convention over Configuration: Ruby on Rails

Another interesting approach to this kind of programming is ruby on rails. Ruby on rails is a framework for the Ruby programming language geared towards web applications [Wik05]. Not only is Ruby a good web application framework but using rails also gives developers access to a great persistence layer called Active Record [THB+05]. What makes this framework a little different from others is that the philosophy behind the framework favors convention over configuration. This means that instead of having to configure mappings in files all over, mapping follows common conventions. These conventions are used to map to tables in a unified way for every application built. As an example one such convention say that the name of a class should directly map to a table in the database which is the plural of the class name. In our User example the convention would then map the User class to the table Users. Rails is however more potent than that, a class named Person will intelligently map to the pluralis People table. Rails will correctly map some irregular word but is of course not omnipotent and it would not be too hard to fool it [THB+05].

The benefits of using Active Record does not stop at mapping classes to tables, since ruby is a dynamically typed language, it is possible to do much more. For instance consider the example below in which we will retrieve the User identified by an Id number 123.

```ruby
class User < ActiveRecord::Base
end

usr = User.find(123)
usr.name = "Stefan Chyssler"
usr.save
```

For someone not familiar with dynamically typed languages the code above may seem strange. The ActiveRecord::Base superclass does not define the name attribute, neither does the class User. Because Ruby is a dynamically typed language it is possible to assign fields to an object at runtime. This enables the framework to load an arbitrary object and assign the fields found in the table to a new instance of an object. The find(123) method recognized the User class as the subtype and will therefore look at the Users table in our database and retrieve the row with primary key 123. To create more sophisticated relations between tables rails use some configuration, but it is minimal to say the least. Readers of this paper should be familiar with the idea that our User is associated with zero or more Orders. Using Active Record to define User and Order classes would then look like:

```ruby
class User < ActiveRecord::Base
  has_many :orders
end

class Order < ActiveRecord::Base
  belongs_to :user
end

usr = User.find(123)
# Do something with each order of the user
usr.orders.each { |n| ... }
```
The declarations in the class does more than just specify relationships between the tables. Active Record will also add methods to the model to help navigate between objects. The belongs_to( ) declaration specifies that the Order class has a child-parent relationship to the User class containing the declaration. Belongs to is the Active Record convention to express that the table that contains a foreign key belongs to the table it is referencing (see schema in previous section). The parent class name is assumed to be the mixed-case, singular form of the attribute name (as belongs_to: user). The foreign key field should be the singular form of the attribute name with _id appended (user_id instead of Purchaser as in database schema). Active Record uses the foreign keys to reference the id columns in the parent tables to retrieve parent instances. Similarly the has_many relation use the plural form class name (see code). It is possible to override these and other assumptions. This is done by passing a hash of options after the association name. The various declaration (belongs_to, has_one and mixed) also control how things get saved and updated, wether it will cascade into child/parents or not. [THB+05]

Following the conventions for Ruby on Rails will almost totally abstract database access away from the developer. Without the need of configuration files the framework has successfully created a very transparent persistency framework. The framework designers however realized that it may not always be possible to follow conventions. The framework does allow developers to issue direct SQL queries, and any inconsistency between classes and conventions can be configured away. In contrast to Hibernate however, most of these configurations are made in the class definition and not in an external configuration file. Both these methods have its own benefits and drawbacks, which will be covered in the discussion section.

When accustomed to these type of conventions it further relieves the developers of redundant work, increasing productivity. Less code to write also means less code to debug and test. Other frameworks that do use configuration also relieves some of the burden by providing "sensible default values" [Hib06a].
2.4 Object Databases

Object oriented programming languages allow programs to be modular, built by objects encapsulating related data. This increases the flexibility and reusability of the program because each objects internal structure is hidden from the user. But this encapsulation imposes a difficulty when it comes to storing and retrieving objects. To successfully store an object we must have access to all information about its current state. If the object is to be stored in a relational database we either need a method that stores the object itself or the object must provide access to all its internal variables. The first alternative is probably the nicest approach but it implies that methods for storing and retrieving must be implemented in each object that should be stored or retrieved; a very time-consuming task and it may not be very efficient.

Fortunately there is another solution: object-oriented database management systems (OODBMS). These databases adhere to the object-oriented ideas such as type hierarchies using inheritance, encapsulation and object referencing. They can store entire objects with a single line of code, no matter how complex the object is. Another great advantage is that they don’t require adaptation of the objects to be stored or retrieved [Wik06a]. The object to be stored is simply passed as a parameter to a save-function of an open database connection.

The following code illustrates how a new item, ASX 2.1 Speakers can be added to an object-oriented database:

```java
Item newItem = new Item("ASX 2.1 Speakers", 395.00);
db.Set(newItem);
```

To retrieve an object a little more effort is required. Object-oriented database systems can usually locate objects using a prototype of the same object. This way of specifying queries is called Query By Example (QBE). By setting properties of the prototype some conditions can be specified that must match the objects to be returned.

A simple query to retrieve all items with the price 395.00 from a web shop could look like this:

```java
Item prototype = new Item("", 395.00);
ObjectSet result = db.Get(prototype);
```

First a prototype object is created where we set the price requirement to be 395.00. Then we pass the prototype to the databases get method and a set of objects is returned.

This technique is very easy to use; however, it has some clear limitations:

- The only supported comparison is equality, ranges cannot be expressed
- AND, OR, NOT conditions are not supported.
- Objects cannot be matched on empty values such as NULL, or 0 since they are interpreted as any value.
• Objects should have constructors that initialize its variables to empty values.

• All object properties are used in each comparison, which is inefficient.

To allow these basic features another query system, called native queries, can be used.

2.4.1 Native Queries

Native queries have evolved from the ideas of safe query objects [CR05] which encapsulates queries in an object-oriented style. The intention of this is to allow programmers to make use of refactoring, auto-completion, syntax- and type checking provided by IDEs and compilers. This is possible because the native queries are expressed with the syntax and semantics of the native programming language. During compile time the queries may be translated into OQL or SQL queries, thereby not introducing any run-time overhead in the application. The actual translation can be done by a separate application or a plug-in to the compiler. The translated queries are sent to the underlying database at run-time. These queries may be string-based, but they are safer then string queries created by a human since they are translated from native queries which have passed type- and syntax checks by the compiler [Cr06].

The native query requesting all items with price up to 395 could look like the follows:

```java
IList items = db.Query(new PriceMax395());

public class PriceMax395 : Predicate
{
    public boolean Match(Item item)
    {
        return item.Price <= 395;
    }
}
```

This shows that the entire query is expressed in the native programming language. The code, however, contains a fault: item.Price should be item.Price. This fault will be caught by the compiler and can easily be corrected. Imagine that a similar fault was introduced in the embedded SQL code of a complex application. It could be very difficult to detect and locate a fault like this.

Instead of scattering object information over several tables as relational databases can do, OODBMSs stores all the information in one location and related objects can be accessed using references or pointers. This implies that when the object is retrieved again from its persistent state all information is there, no time-consuming joins are needed.
3 Design Pattern: Data Access Objects

Similar to the Bridge Pattern [GHJV95], the DAO pattern make use of interfaces which in turn are then implemented by some module to work with data sources. The data source could be a persistent store like an RDBMS, an external service, object-database or whatever the implementation chooses. The client-application relies on the DAO implementation and uses the simpler interface exposed by the DAO. The DAO thus completely hides the data source implementation details from its clients. Because the interface exposed by the DAO to clients does not change when the underlying data source implementation changes, this pattern allows the DAO to adapt to different storage schemes without affecting its clients or business components [jav06].

Using the DAO pattern makes it easier for an application to migrate to a different database implementation. This is possible because the application have no knowledge of the underlying implementation. Migration from one implementation to another would then involve changes only to the DAO layer. If used together with the factory pattern [GHJV95] this can be as easy as changing one line of code to completely use another data storage strategy.

The DAO design pattern also reduce complexity in the persistent objects since it is the DAO that manages all data access, business objects can then focus on their business logic without the need to know how to store themself. Eventual SQL statements are stored within the DAO implementation, increasing the readability of the business objects. However this pattern may break encapsulation, as the DAO pattern may need access to properties to store and retrieve object data.

A drawback of the DAO pattern is that it adds another layer of objects between the client and the data source that need to be designed, implemented and maintained. The benefits however is often worth the additional effort, the centralization of data access into this a separate layer makes the application easier to maintain [jav06].
4 Discussion

The choice of persistence technique should always be based on the needs and requirements of the application. The binary files-technique is an ad-hoc solution that may work well for small and specific amounts of data but become time-consuming and difficult to maintain as the application evolves. Binary files are well suited for large pieces of stream-like data such as music or videos. It is less suitable for high concurrency systems such as web-shops. Relational databases however are well suited for web-based applications, for instance the LAMP (Linux Apache MySQL PHP) package is a popular example in which the GNU licenced MySQL DBMS is used. When using relational databases however, developers face alot of tedious work with mapping results back and forth. It is the mapping which tools such as Hibernate and Active Record takes care of, with a penalty to execution performance. Despite this performance penalty, these tools together with a capable RDBMS are very well suited for developing high concurrency systems such as a web-shop. The difference between these two tools are that Hibernate favours externalized configuration whereas Active Records favours in-code configuration(or no configuration), both has its advantages and disadvantages. The former makes it easy to change configuration without source-code access, whereas the latter makes it easier for the developer to have control and reduces information redundancy and is less fragile to refactoring.

While object-oriented databases are good at fast retrieval of complex single objects since that is what they initially was designed for. They also provide a close mapping between the database system and object-oriented programming languages. Often these databases are tightly connected to one or more object-oriented programming languages. While this may be okay for some systems, it does not permit the application-data to evolve outside the programming language.

The data access object pattern makes it easier for an application to migrate to a different database implementation, which in turn makes it more manageable. DAO may even be used to speed up the application development using a test-driven design. The DAO interfaces can then be used to provide mock-objects. Mock-objects are dummy objects that simulates real implementations, allowing an object to be tested in isolation[FMPW04]. DAO is in most cases a good pattern and it’s use should be encouraged, but it may be overkill for prototype applications.
References


