Persistence Options for Object-Oriented Programs

Wolfgang Keller♣
Liebigstr. 3; 82166 Lochham, Germany
EMail: wk@objectarchitects.de
http://www.objectarchitects.de/

1. Abstract

This paper is an extended abstract for a talk at OOP 2004 on “Persistence Options for Object-Oriented Programs”. It gives you a bird’s eye view of the field of persistence options for o-o programs with a deeper look at object/relational access layers. You will get an idea of an ideal persistent o-o language and the ODMG standard. This will lead to forces which you need to know when deciding about persistence options.

You will look at various options like flat files, object databases, relational databases, o/r databases, and finally at how o/r access layers for relational databases can be built.

Starting with application types we move on to a rough overview of how to handle persistence in business information systems. We will come by the mechanisms needed for build-your-own cheap solutions to heavier ones using meta-data. The paper ends by showing you what to expect in environments like J2EE/EJB and .NET.

2. Who Will Profit from this Paper and How

If you are an open source developer of persistence layers or a senior software architect who hibernates on the Server Side or sleeps at the Java Lobby then this paper will bore you. This paper is meant as an introductory article that shows developers what factors to look at, when tackling persistence. Senior developers may also profit as they might draw a few new aspects from the paper – especially the application styles in section 5. They might also profit as this paper is suited to spoil their appetite for a “roll your own” persistence solution. Software managers will get the opportunity to ask their tech people a lot of nasty questions if they enter the door with the idea to build their own persistence solution. In most occasions you will find ready made solutions that do the job.

♣ this paper is not written under the logo of my employer AMB Generali Informatik, as the topic has not got much to do with my daily job obligations there. Hence this paper is a personal statement and all opinions expressed herein are my personal ones. Nevertheless I’d like to express my thanks to AMB Generali Informatik for letting me participate in OOP 2004. The technical information in this paper has been compiled with professional diligence. But errors can happen and will happen. The author therefore rejects any responsibility for the topicality, correctness, completeness or quality of the information provided. Liability claims regarding damage caused by the use of any information provided, including any kind of information which is incomplete or incorrect, will be rejected.

3. This Paper in the Context of Other Writings on O-O Persistence

A lot has been written about object persistence. Before we define the term “object persistence” for those who can use an idea or a repetition, we will put this paper in a context of other writings on object persistence so that those of you who know the field can get an idea whether it is worth for you to read on or not.

- **Applications Styles and Persistence:** Most papers on persistence deal with persistence for business information systems. They deal with the question ”how to design a persistence layer for a business information system” and assume that you will use a relational database like most business information systems use a relational database nowadays. This paper starts a few questions earlier by asking “what is a good persistence style for a given application style”. Because you could also use stream persistence, xml-files, object databases, and object/relational database extensions to name a few. Literature on the question “what is the right persistence mechanism for what application style” is somewhat less widespread. We will cite some later in the chapter and the chapter will be somewhat deeper than the other ones.

- **O/R Mappers Explained:** Once you have checked that a so called object/relational access layer is suitable for your application style, it is good to know how such “beasts” are built. Let us use a little analogy here: In general you don’t need to know how your car is built, but if you come to a critical situation it is good to know at least a portion of what is going on below the hood. This doesn’t mean that you want to become an engineer in car manufacturing – but you might be a better driver if you know a bit more. Hence the most important key message of the chapter is that you should not build an o/r access layer unless you want to specialize in it as a software tool producer or want to become an open source programmer for such solutions. But it is good to know some. This chapter will give you some insight and direct you to the literature that can give you insight down to the level that is needed by those who engineer such layers.

  There is plenty of literature on O/R mappers and a paper like this is not suited to beat those in the level of detail. But we can give you an overview.

  - If you want to built a business information system and if you are a developer you might want to read a “how to” book. These are available as manuals of common O/R mappers (e.g. [TOP2003]) or also as books on standards like the JDO-standard [JoRu2003, Roos2002] or open source initiatives like Hibernate [BaKi2004].
  - If the detail in these books is not enough for you, you can dig one level deeper into the patterns literature on o/r access layers. In the meantime a lot has been consolidated by Martin Fowler et al. in their Patterns of Enterprise Application Architecture [Fow2002]. You will find more and also lots of free material at Scott Amblers website http://www.agiledata.org/ and also at my site http://www.objectarchitects.de/

- **Persistence in EJBs:** is a field which we will only touch. In general EJBs come with their own persistence models called Bean Managed Persistence (BMP) and also Container Managed Persistence (CMP). The number of books on EJBs is quite countless. You will find EJB "how-to" books, EJB patterns books and more. As EJBs are complex it is good to read what people do wrong when using EJBs and especially also when persisting EJBs. These topics can be found in a book called "Bitter EJB" [Tate+2003].

- **A Few Remarks on Persistence in .NET**: At the end of the paper we will have a look at the state of the art for persistence in .NET. You will find almost everything there which you know from the Smalltalk, C++ and Java Worlds. All you have to get used to are maybe other names for the same patterns and principles.

As you might have noticed, this is quite an "Omnibus" – which means that if you are looking for the deepest in technical insight, you should skip it and turn to the detailed original literature. But sometimes there are also people who need an introduction to the field and a survey. This is what the talk and the paper are intended for.

4. What is Persistence Anyway?

A Basic Definition

Persistence is the ability of an object to survive the lifecycle of the process, in which it resides
Objects that „die" with the end of a process are called transient

This definition is rather basic and it implies a few things concerning the relation between databases and persistence: There can be very primitive persistence mechanisms which do by far not rely on a database. E.g., if at the end of a process you file all objects to a flat file or an XML file, than this is also persistence.

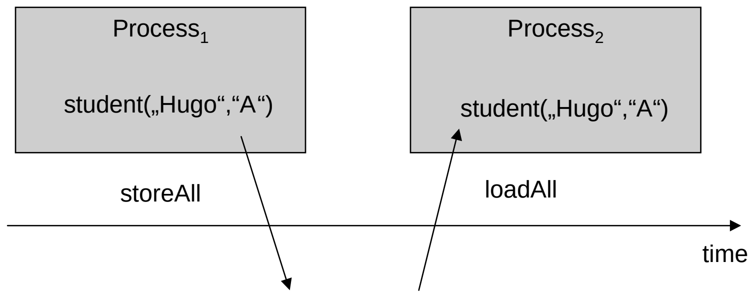


Figure 1: On termination of Process$_1$ all objects, e.g. student(„Hugo", ..) are externalized to some kind of storage. When Process$_2$ starts, all objects are loaded again. This is some primitive form of object persistence.

Different Options to Implement Persistence

In practice streaming all objects to some flat file is only one of many options to implement persistence. For some application styles this does the job. Often e.g. "Concurrency" and "Isolated Transactions" are a requirement when multiple users access the same set of data from different processes and hence database features are needed:

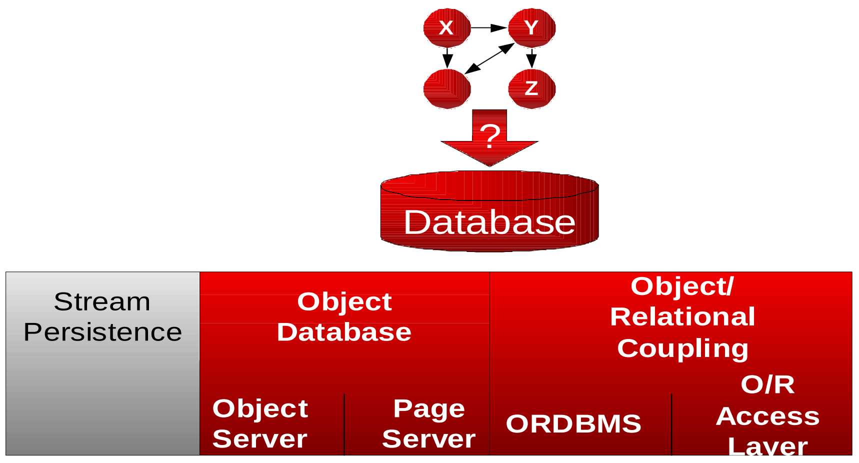


Figure 2: Persistence Options: Where to store your objects

If it comes to storing your objects in a database instead of in some form of file you can choose between object databases and relational databases.

- Object databases come mostly in two flavors: As page servers and object servers. For the differences see e.g. the tutorial by Jens Coldewey [Col1998] which is based on material by Akmal Chaudhri[1]. We will revisit OODBMS if it comes to orthogonally persistent systems.

- Relational databases as a storage medium for objects require a larger transformation mechanism as objects have a lot of properties, which are not present in the relational mechanism. For those databases, which implement the so called object/relational model, this layer is somewhat thinner, but still thick enough[2]. The fact that objects and relational databases have quite different underlying meta-models is called the “The Object-Relational Impedance Mismatch”. We’ll come back to that later once we have explained orthogonal persistence.

Transparent (or Orthogonal) Persistence and the OODBMS Manifesto

The idea of orthogonal persistence (also called transparent persistence) is that at the coding level you deal with a system that knows objects only and that you should not need to be very aware of the fact that some objects are persistent (and some others are transitive).

In the late 1980ties a group of researches stated the so called “*The Object-Oriented*

[1] Akmal Chaudhri used to have an excellent web site on ODBMS performance benchmarks (http://www.soi.city.ac.uk/~akmal/) which, among other things, explained a lot about the differences of e.g. page servers and object servers. Hope he will put it on the web again.

[2] Besides there’s not much practical experience with access layers that map objects to a object relational database. There exist far more products that map objects to “pure” relational databases.

Database System Manifesto" [Atk+1989] which described what an orthogonally persistent database for objects should look like. We have extracted the properties in a table and compared them to the expressive means of o-o programming languages and relational databases.

From this you will clearly see that there is a mismatch between objects and relational databases:

OODMS Manifesto: **Mandatory Features**	**Object/Relational Access Layers:** **Covered by**
(1) Complex Objects	Your programming language for business objects (like C++, Smalltalk, Java, ...), your RDBMS plus an access layer. RDBMS do not provide mechanisms for complex objects.
(2) Object Identity	In o-o languages provided via the memory location of an object (object's address). Translated to databases see Object Identity Pattern [Fow2002]
(3) Encapsulation	Your programming language.
(4) Types and Classes	Your programming language
(5) Class or Type Hierarchies	Your programming language plus patterns for **Mapping Objects to Tables**.
(6) Overriding, overloading and late binding	Your programming language
(7) Computational Completeness	Your programming language
(8) Extensibility	Your programming language plus patterns for **Mapping Objects to Tables**. – see section 6
(9) Persistence	Whole access layer plus relational database (RDBMS).
(10) Secondary storage management	RDBMS
(11) Concurrency	RDBMS plus patterns for transaction control and locking strategies.
(12) Recovery	RDBMS
(13) Ad Hoc Query Facility	access layer on top of RDBMS

Table 1: Core responsibilities of an Object-oriented Database Management System

Most of the functionality listed in Table 1 comes with your object-oriented programming language (like 1, 3, 4, 5, 6, 7, and 8). The challenge is to make your object-oriented programming language's objects persistent, giving them the ability to survive the termination of the actual process and to be used again in other (also in parallel) processes.

Therefore the other requirements are typical requirements that you find for databases (like 9, 10, 11, 12,and 13). See any database book for an explanation, e.g. [Dat2003].

As code says more than many words let's have a look at the ODMG Version 3.0 Java interface for persistent objects. You will see that persistence is far from hidden here, which is also a main point of critique from the advocates of "lighter" persistence frameworks like Hibernate [BaKi2004].

```
public static Product findProductByName(String name) throws Exception
    {
        Implementation impl = OJB.getInstance();
        Transaction tx = impl.newTransaction();
        tx.begin();

        OQLQuery query = impl.newOQLQuery();
        query.create("select products from "
                     + Product.class.getName()
                     + " where name = $1");
        query.bind(name);
        DList results = (DList) query.execute();
        Product product = (Product) results.iterator().next();

        tx.commit();
        return product;
    }
```

Figure 3: Code snippet that shows how a Product.findProductByName – method is implemented using the ODMG 3.0 interface. This looks much like an SQL binding on another "more object-oriented" level". This is also why critics would describe such interfaces as far from transparent. On the other hand it will stay very hard to hide the complexity of concurrency and transactions completely from the user of a persistent o-o language (Source: http://db.apache.org/ojb/odmg-tutorial.html)

Section Summary

You have seen that the concept of persistence *"per se"* is a quite simple one. Things become more complex and options become numerous once you access the same set of objects in a database fashion from numerous parallel processes. This implies in most cases that you will use a database as your underlying persistence mechanism. This could be an object database or a relational database. If you go for a relational database you need something that bridges the gap between an orthogonally persistent language (as defined e.g. by the OODBMS manifesto) and the constructs offered by a relational database.

The next question is when to use which persistence option. This has a lot to do with application styles.

5. Application Styles

If we talk about application styles we do this in order to make you see a few patterns in persistence usage that allow you to use the cheapest and most efficient solution for your application's persistence problem. We will give you a list of criteria – the pattern people would call them forces – that allow you to better judge the dimension of your challenge and that will determine the "Gestalt" of your solution.

Important Forces

- **Number of users:** The number of users is a sometimes ignored force in persistence design. From both sides. People who are used to build database oriented solutions tend to ignore the fact that a system they build this time might only have one or a few users and others forget to make the point that a system for one user will not scale for hundreds or thousands of users.
    - **Single User Systems:** Can often be found in design and programming. Ever thought of storing e.g. you program in a relational database instead of in a program (flat) file. Don't worry. It has been attempted to store programs line by line and statement by statement first in relational databases and as this did not work in object databases. As you might guess this was far from a success. The simplest thing that works here is still good old flat file persistence. But observe the size of the objects.

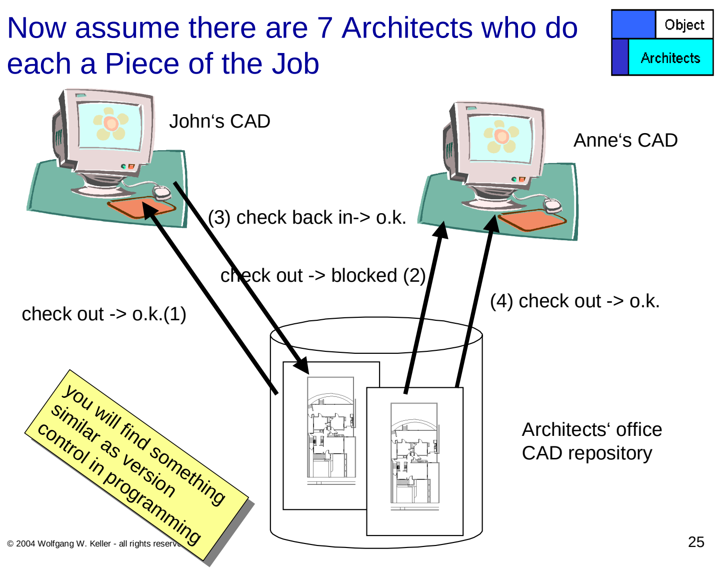

Now assume there are 7 Architects who do each a Piece of the Job
Object
Architects
John's CAD
Anne's CAD
(3) check back in-> o.k.
check out -> blocked (2)
(4) check out -> o.k.
check out -> o.k.(1)
you will find something similar as version control in programming
Architects' office CAD repository
© 2004 Wolfgang W. Keller - all rights reserved
25

Figure 4: Check - in / check - out persistence in design tasks. A group of people is working on a set of models. Each team member works on one artefact at a time. You find this in programming, design and similar engineering tasks.

  - **Few Users / Check in – Check out:** There's kind of a pattern that assigns relatively small amounts of data (like programs) to so called check in / check out persistence (see Figure 1). Think of a programming or design project. One guy works on one artifact at a time. Checking a program file out of a PVCS repository, working on it and checking it back in later is an example for this.

  - **Mass Users / Fine grained**: Order processing or airline booking is an example where

    - many users
    - access a vast total amount of data
    - each using only a small amount of data
    - with a high likelihood of collisions (two people want to book a seat on the same plane at a time)

    Such a scenario smells a lot like using some kind of database.

- **Overall size of the data to be stored:** If you measure your data in a few megabytes then this is something that a single personal computer can hold in its working storage. If you measure your stuff in gigabytes or terabytes, then you will use some kind of secondary storage management where only a small fraction of your total data resides in your client's working storage while the rest resides in some kind of database. The single user design app as described above is a case where everything may reside in your client's memory. The airline booking system with terabytes of data calls for some kind of database.

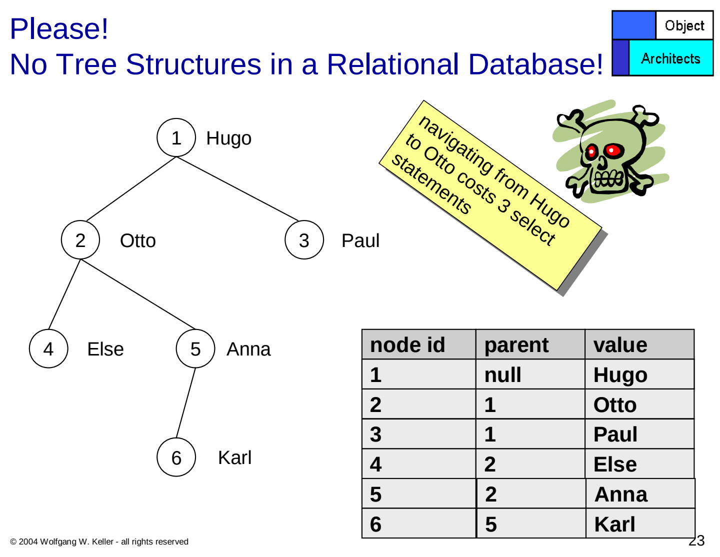
- **Concurrency and likelihood of collisions:** With an engineering design job it is not very likely that it makes much sense, if two engineers work on the same model at the same time[3]. If this makes sense, then each one has a local copy and models need to be merged later. For the above quoted booking system the likelihood of collisions is very high, as two or more people trying to book seats on the same plane is the default case. In this case you need a persistence mechanism that can handle concurrency, isolation and hence so called ACID transactions. Most databases implement such mechanisms – as these are the main distinctive features of databases versus other file systems.

- **Structure of the objects stored**: Imagine a table of orders and another table of order positions. This is one of the standard examples if you teach relational databases. But not all data are organized in tables, having a primary key and are accessed in a random fashion.

  Another very wide spread kind of data structures are all kinds of trees, graphs and networks. You find them especially in engineering and design. The access

[3] Even "Pair Programming" is done with two programmers working on one image (design).

here is seldom ever random but more often “navigational” – you navigate from one node of the tree to a neighboring one. Try this with a relational database. Figure 5 gives you an impression of what to expect.

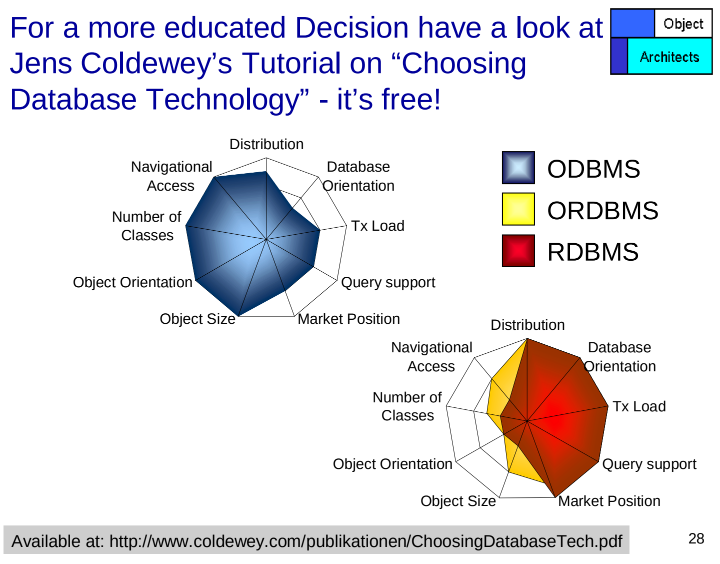


Please!
No Tree Structures in a Relational Database!

Object
Architects
1 Hugo
2 Otto
3 Paul
4 Else
5 Anna
6 Karl
navigating from Hugo to Otto costs 3 select statements
© 2004 Wolfgang W. Keller - all rights reserved
23

node id	parent	value
1	null	Hugo
2	1	Otto
3	1	Paul
4	2	Else
5	2	Anna
6	5	Karl

Figure 5: Relational databases are not best suited for large tree and network data structures. You can find easy workarounds as long as you are able to load all the data in your working storage (as the above Figure might suggest a bit). But you might find yourself in trouble once you have gigabytes or terabytes of such structures using an RDBMS.



For a more educated Decision have a look at Jens Coldewey’s Tutorial on “Choosing Database Technology” - it’s free!

Available at: http://www.coldewey.com/publikationen/ChoosingDatabaseTech.pdf

Figure 6: A few more technical criteria and how different kinds of databases fulfill the respective needs. Source [Col1998]

- **More technical criteria:** There are far more technical criteria that describe the structure of the objects you consider to store. Have a look at the Tutorial by Jens Coldewey [Col1998] that will give a more detailed account of such properties and see Figure 6 for an idea.

The following two forces are not application styles but are nevertheless important:

- **Company Politics:** Once you have decided that a database is the tool to use the question will be “which database?”. Most large organizations have their favorite (relational) database system in place – which makes much sense as you will draw significant economies of scale if you use only one type of system in a large organization. But on the other hand a page server o-o database might be the ideal solution for your problem. In this case you need to make the conflict explicit and you need to push a solution that does not kill your project. All too often, the big iron wins and the small project that does not have the 98% profile suited for the standard solution fails.

- **Vendor stability:** A force often neglected by techies is vendor stability. We look too much at features and slick technology and often neglect the commercial aspect to ask whether the company is going to be around in 6 months time. Assume you work for a Fortune 500 company. What sense does it make if you select a database system (or persistence mechanism) from a supplier that has let’s say € 3 Mio. turnover and a yearly cash burn rate of also € 3 Mio. If the venture capitalist behind that company decides that the company is a failure, you are left on the street without tech support. Not a thrilling idea for a Fortune 500 company. Unfortunately this problem exists for many of the OODBMS vendors. Table 2 gives you an idea of what to expect when you want to fight for the use of an OODBMS in a larger organization. The total market share of all OODBMS vendors taken together is in the 0.5 percent range and still shrinking.

Category	**2002**	**2001**	**Market Share (%) 2002**	**Market Share (%) 2001**	**Growth Rate (%) 2001-2002**
Desktop DBMS	455	502	5	6	-9
Object DBMS	42	54	1	1	-22
Pre-relational DBMS	1,225	1,228	15	14	0
RDBMS	6,642	7,139	79	80	-7
Total DBMS	**8,363**	**8,922**	**100**	**100**	**-6**

Table 2: Worldwide Database Revenue by Category for 2000-2001 Based on New License Sales (Millions of Dollars). Source: Gartner Dataquest (June 2003).

More forces: If you design an access layer there are far more forces involved. For an account see e.g. [Kel1998b].

Section Summary: A Few Rules of Thumb

This section demonstrated that it is important to know what type of application you are facing in order to build the most appropriate persistence solution. The following sums up a few rules of thumb extracted from the above. Please note that due to collision of forces the below rules might also collide – this is all too natural in software design:

You have high concurrency - many users working
on the same data
… Use a „real“ database like a relational or object database[4]

You need „true“ database features like recovery,
logging, concurrency
… Use a relational or object database :-).

Your amount of user data is several times larger
than the working storage of you computer
… Use a relational or object database.

Your amount of user data is small compared to your
computer size, concurrency is low to non existent,
the problem is a check in / check out problem
… Consider using stream persistence.

You build an Enterprise Information System like
order entry, bookkeeping and the like
… Do what everybody does - use a RDBMS

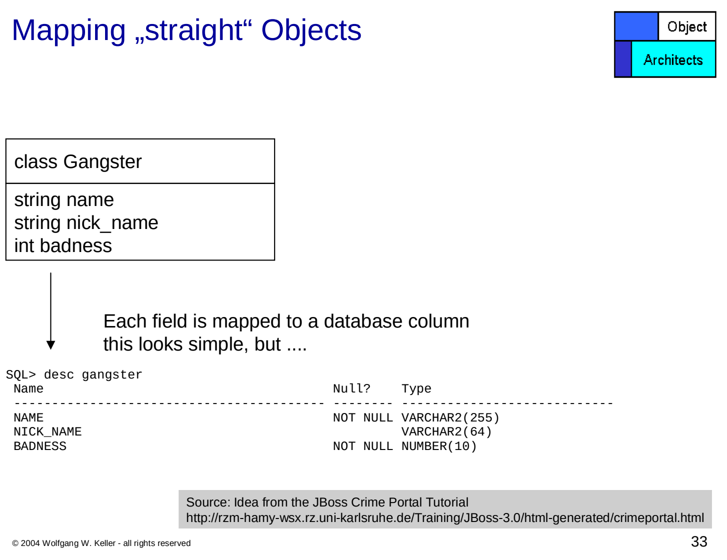
6. O/R Mappers Explained

You are now in a situation that will be the situation of many of your fellow programmers and fellow software architects. You have decided to use a relational database as the storage backend for your persistent data and you need to understand how all these object/relational mapping tools work. This chapter will give you a crash course in access layers. Be aware that this chapter cannot substitute deeper study of the literature cited in section 2.

4 And don’t forget that object databases have a very very low market share.

Mapping Straight Objects

Let’s start with the presumably easiest thing that can be done[5], mapping an object with only a few dumb attributes to a relational database. Figure 7 shows a very simple mapping. A class *Gangster* is mapped to a database table.



Mapping „straight“ Objects

Object
Architects

class Gangster

string name
string nick_name
int badness

Each field is mapped to a database column
this looks simple, but ....

```
SQL> desc gangster
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 NAME                                      NOT NULL VARCHAR2(255)
 NICK_NAME                                          VARCHAR2(64)
 BADNESS                                   NOT NULL NUMBER(10)
```

Source: Idea from the JBoss Crime Portal Tutorial
http://rzm-hamy-wsx.rz.uni-karlsruhe.de/Training/JBoss-3.0/html-generated/crimeportal.html

© 2004 Wolfgang W. Keller - all rights reserved 33

Figure 7: Mapping a simple class to a simple database table

Here we have only simple types as attributes and everything looks straightforward. If you do practical mapping, then you will come across some of the following questions sooner or later:

- How do you map variable length data types like e.g. strings?
    - Lots of VARCHARS do not speed up a database!
- ENUMS need special treatment.
- SQL data types are not semantically identical with e.g. Java data types.
- How do you deal with aggregated complex types like e.g. records?
    - Put them in an extra table (slower).
    - Unfold them into various tables (tougher to maintain).

In many cases the answers will be vendor specific, as even standards like ODMG or JDO only guarantee an interface but do not prescribe a standard mapping

The CRUD Pattern

Now that we know, how the simplest of objects can be mapped to a database, we can

[5] As a running example we will use an example inspired by the Crime Portal – a JBoss demo application. Thanks to the JBoss Community for the idea.

look at the cycle how these objects are **C**reated, **R**ead from the database, **U**pdated in the database and **D**eleted from the database. If you combine the letters you have the acronym CRUD which could be called "the mother of all access layer patterns".

Let's have a look at the mechanics in the order given above. The caption for Figure 8 explain enough about what happens.

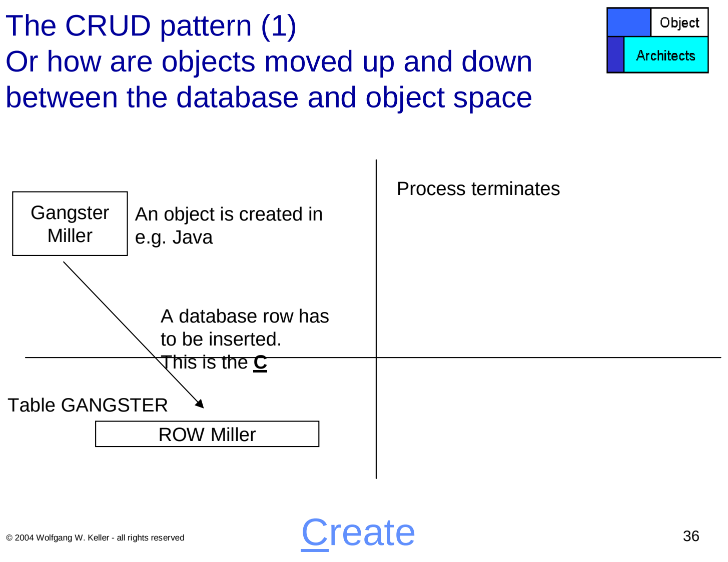

The CRUD pattern (1)
Or how are objects moved up and down between the database and object space
Object
Architects
Gangster Miller
An object is created in e.g. Java
Process terminates
A database row has to be inserted. This is the C
Table GANGSTER
ROW Miller
© 2004 Wolfgang W. Keller - all rights reserved
Create
36

Figure 8: Objects are first created in memory. Whenever a transaction ends, they are written to persistent storage, e.g. a database. This "insert statement" is the C in CRUD.

Next come the R and U. This is a normal cycle, where an object is loaded (**R**ead) by name, modified and then rewritten to the database using an **U**pdate statement. Figure 9 shows enough. If you want more sample code have a look at the Reflective CRUD pattern (http://www.inf-cr.uclm.es/www/mpolo/yet/ [Pol+2001]). You will find plenty of sample code there.

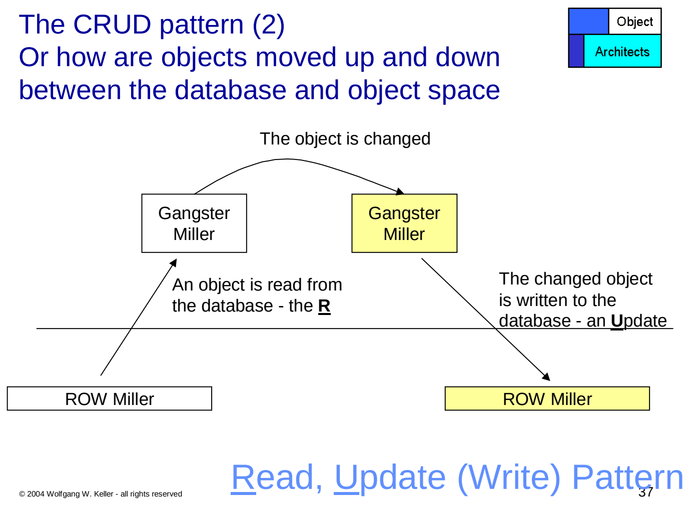
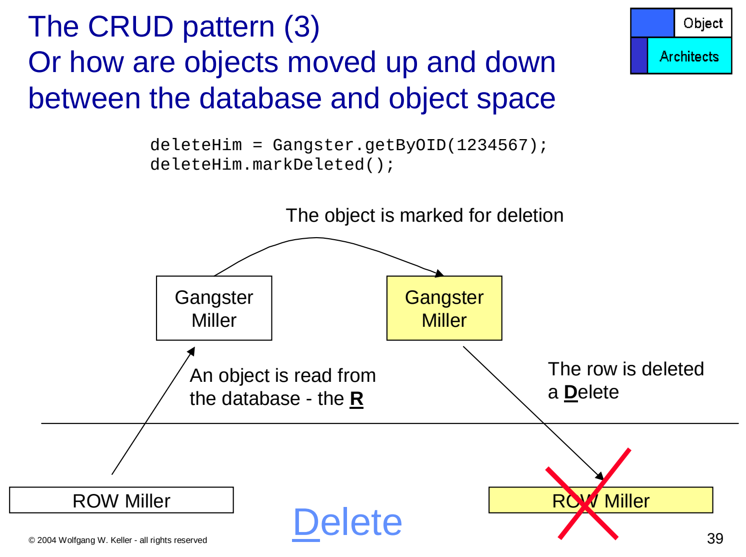

The CRUD pattern (2)
Or how are objects moved up and down between the database and object space
Object
Architects
The object is changed
Gangster Miller
Gangster Miller
An object is read from the database - the R
The changed object is written to the database - an Update
ROW Miller
ROW Miller
© 2004 Wolfgang W. Keller - all rights reserved
Read, Update (Write) Pattern
37

Figure 9: Reading an object, modifying it and writing it down to the database using an update statement.

What remains is the question how to **D**elete an object also from the database. In a fully transparent persistent language you would expect that this happens during garbage collection. In practice it is not very handy to have a garbage collector work on a giant database. Hence the task is delegated to the programmer who has to state explicitly that the object shall be deleted on end of the transaction or process. Figure 10 will show how this is done.


The CRUD pattern (3)
Or how are objects moved up and down between the database and object space

```
deleteHim = Gangster.getByOID(1234567);
deleteHim.markDeleted();
```

Object
Architects
The object is marked for deletion
Gangster Miller
Gangster Miller
An object is read from the database - the **R**
The row is deleted a **D**elete
ROW Miller
ROW Miller
Delete
© 2004 Wolfgang W. Keller - all rights reserved
39

Figure 10: an access layer will issue delete statements for all objects that have been marked for deletion by the application program above the access layer.

Object Identity and the Identity Cache

Much has been written about Object Identity. One of the better accounts on the topic can be found in the upcoming book "Hibernate in Action" [BaKi2004]. In this short article we will demonstrate the problem in a brief "hands-on" fashion and we'll also show the solution common to most mapping layers. For more detail have a look at the "Object Identity" Pattern in [Fow2002]. But now let's move on to our code example. Look at the following piece of code:

```
Gangster bigBoss;
Gangster capo;
Gangster arrestHim;

capo = new Gangster(„Miller“,“the Killer“,9);
bigBoss = new Gangster(„Miller“,„the Smart“,13);

arrestHim = Gangster.getByName(„Miller“); // ????
```

Here we create two memory instances of class Gangster. So for the programming language we have two different objects. The question now is what we want as application semantics for the class. We have created two instances using the same name "Miller". If the name was the primary key in the database, we would run into trouble at the moment the magic below the programming language layer tries to write the second instance to the database. Hence we have to be very careful with so called object identity and different notions of it in the database and at the programming language level.

The solution for this is an Identity Cache. The following piece of code and Figure 11 will demonstrate the mechanics.

```
Gangster capoDeiCapi;
Gangster ilSoloCapo;

capoDeiCapi = Gangster.getByName(„Corleone“, „Vito“); // (1)
ilSoloCapo = Gangster.getByName(„Corleone“, „Vito“); // (2)

capoDeiCapi.setBadness(MAXBADNESS);
if (MAXBADNESS != ilSoloCapo.getBadness()) {
     // Palermo!!! – we’ve got a problem
};
```

Here we have two variables that should reference the same object instance, “Don Vito Corleone”. If the access layer did a bad job it would load two object instances in memory from the same data in the database. In order to prevent this from happening we need to have a mechanism that checks whether the object is already loaded in memory from the database.

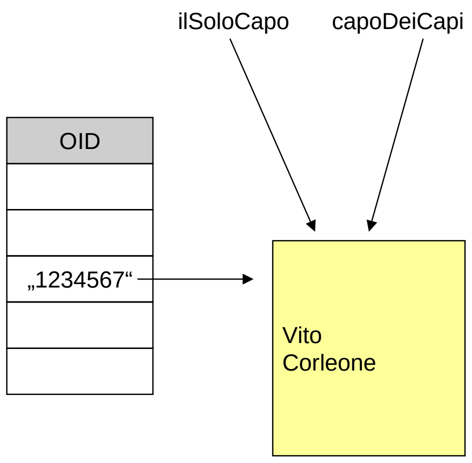


Figure 11: Identity Cache. Two variables are referencing the same memory object instance which is identified by an “artificial” object ID. This artificial object ID is also present in the database table for the Gangster object and is used to prevent multiple memory instances of the same object within the context of a single process and transaction. An identity cache is typically implemented as a hash table using the object ID as the key.

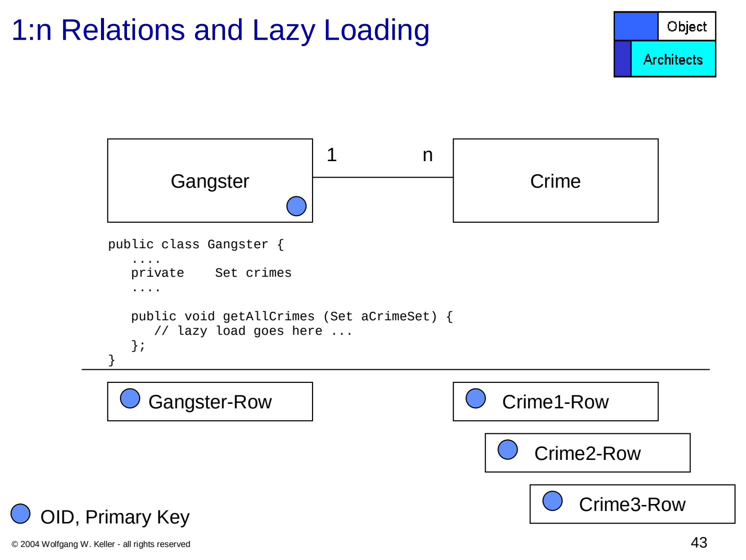
The first time the *getByName* method is invoked, nothing can go wrong. The second time, the *getByName* method needs to check, whether the Gangster with the object ID of “Don Vito Corleone” is already loaded in memory. In this case the *getByName* method will just return a reference to the memory object instance already created by the first call or any other access layer mechanism before. The way this is implemented may vary[6] a bit. But the Identity Cache needs to be used in any case.

1:n Relations and Lazy Loading

Two more questions for access layers are: How do you map 1:n relations and how do you handle loading networks of objects. Figure 12 can be used first to demonstrate how 1:n relations are mapped. One *Gangster* can commit n *Crimes* (1:n relation). This is mapped to the database by giving each *Crime* record in the database the primary key (object ID) of the *Gangster* by whom it was committed.

The next problem is whether you always want to load all crimes if you load a Gangster. The answer is most likely NO because relation networks tend to be more complex and relation chains tend to be longer than in this example. A *Gangster* has committed crimes. A *Person* may be a victim of the *Crime*. Other Persons may have been *Witnesses* and so on and so on. Now imagine that by default you load every object that as some transitive relation with the first object that you touch. In the case of large databases this will blow your working storage or in the best case be a performance problem.

The solution to this problem is known as lazy loading and is implemented by some form of Proxy Objects that trigger a load when they are touched or also Smart Pointers as they are called in C++.



```
public class Gangster {
   ....
   private    Set crimes
   ....

   public void getAllCrimes (Set aCrimeSet) {
      // lazy load goes here ...
   };
}
```

Figure 12: Demonstrates how 1:n relations are treated.

For the sake of completeness it is important to note that sometimes also prefetching can be a much desired behavior. When you know that you need all *Crimes* each time you touch a *Gangster* object you may want to have a mechanism that prefetches what

6 go to the database, get the object ID and the data for Vito Corleone and check whether he has been loaded by comparing the object ID against what is already in the map. The other variant is check the objects reachable in the Identity Cache by content and go only to the database if the object cannot be found in main memory.

you need with one roundtrip to the database.

Persistence with Less Programming: Exploiting Meta Information

If you look at the actual code that reads objects from databases and writes them back down to them, you will soon find this code to be pretty schematic and boring. Whenever programmers find something boring to write, they tend to automate the process. If you look at the following code snippet you will find, that much of the code is just a repetitive frame and the rest can be generated from knowledge about the class enhanced with the knowledge about what attributes in the database the attributes in the class can be mapped to (marked red).

```
private void storeRow() throws SQLException {
      String updateStatement =
            "update GANGSTER set NAME =  ? ," +
            "NICK_NAME = ? , BADNESS = ? " +
            "where OID = ?";
      PreparedStatement prepStmt =
            con.prepareStatement(updateStatement);
      prepStmt.setString(1, name);
      prepStmt.setString(2, nick_name);
      prepStmt.setDouble(3, badness);
      prepStmt.setString(4, oid);
      int rowCount = prepStmt.executeUpdate();
      prepStmt.close();
      if (rowCount == 0) {
         throw new EJBException("Storing row for id " + oid + " failed.");
      }
   }
```

Use of meta information in persistence frameworks comes in many flavors:

- In EJB-CMP you write huge XML files that contain the mapping information.
- In JDO most of the meta information is derived via analysis of the Java Byte Code
- Other mappers use the Java Reflection API

This section is just a first pointer to the subject. There are lots of discussions but there are no documented patterns yet on how people use meta information and what is good and bad with respect to which forces.

Mapping Inheritance and Polymorphism

Again for this topic also there's ample literature. You will find the same ideas about mapping objects to tables in “Patterns of Enterprise Application Architecture” [Fow2002], and also in a paper called “Mapping Objects to Tables” [Kel1997] and on even more spots on the web.

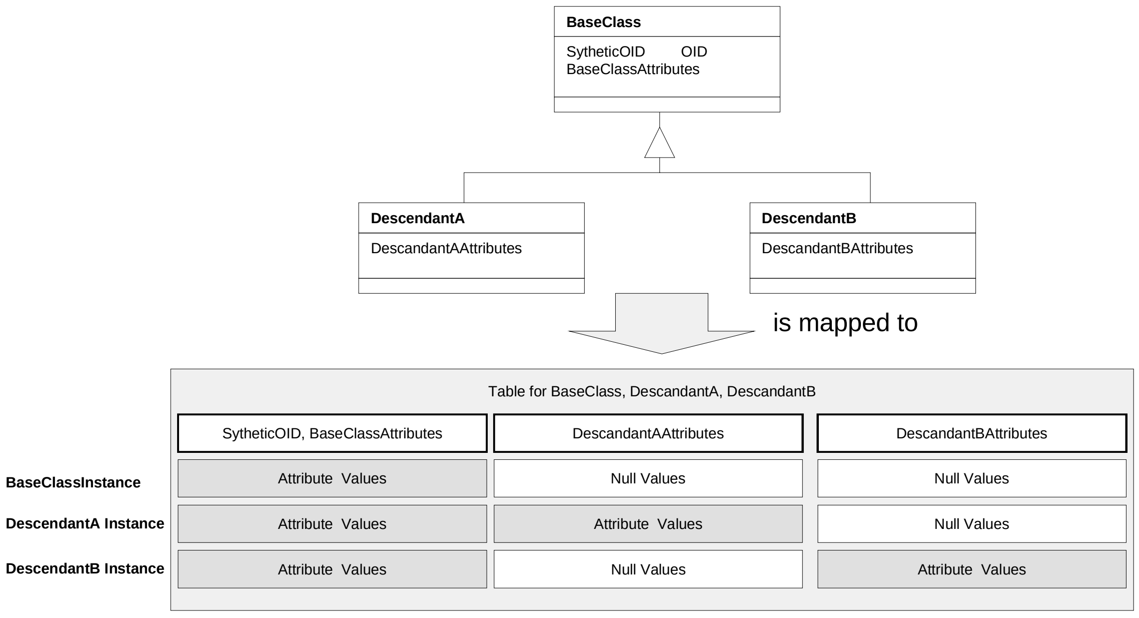


Figure 13: One Variant of Mapping Inheritance: This one maps all classes of a class hierarchy to a single table. The advantage is that polymorphic queries are easy to execute here. The drawback is excess consumption of space for Null values. (For more variants and a comparison see [Kel1997]

Hence this paper is not a suitable spot to condense a 40 pages subject to a half page. We will only demonstrate one aspect which is somewhat important. You are given an idea of how inheritance is mapped. Figure 13 shows one variant of how to map inheritance. A whole class hierarchy is mapped to a single table. This is very valuable if you have lots of queries who search for all kinds of descendants in the hierarchy – so called polymorphic queries. It is less favorable if you have to optimize for space. There are more variants like mapping each class to its own table. With this you are very slow if you are manipulating descendants deep down in the class tree, because you have to read, insert and update all tables along your tree path – but it is acceptable for polymorphic queries. This gives you an idea that there is more to read. With the above pointers you can find a book version [Fow2002, chapter 12] and also a free web version [Kel1997] of the information.

Transactions

Transactions are another field to consider in access layers. In order to get an understanding of how to handle them one would have to explain here

- The lost update problem and isolation: Why do you need so called ACID properties after all? For this aspect consult a standard database book like [Dat2003]
- Optimistic versus pessimistic locking and when to use what. This is tougher to find, because the transaction classics see optimistic locking as a No-No while the pragmatics use it all the time. For a free and brief explanation see [Thor2001] of see the Optimistic Offline Lock Pattern [Fow2002, page 416ff]
- How transactions are implemented in an access layer. This is easier to find as

the Transaction Object Pattern. See e.g. [Fow2002, Kel+1998a].

The sum of all this would result in a 40 page book chapter and would again overstress the extent of this extended abstract.

7. Persistence in EJBs

Persistence in EJBs is not a box of chocolates. EJBs by themselves are not what one would call easy to understand. With the transition from EJB 1.1 to EJB 2.0 a lot has changed in the persistence models so you need to be aware what version you are talking about. For the following we assume EJB 2.0. And there are also a lot of discussions about what is good EJB implementation style and when and whether to use EJBs at all. One of the better books on the subject is Bitter EJB [Tate+2003].

Figure 14 gives you an overview of persistence options for EJBs. The first choice you have to make is whether you want to rely on what EJBs provide (left branch) or whether you want to rely on other, often lighter mechanisms.

EJBs were designed with a model in mind that allowed you distribute beans across different Java Virtual Machines on arbitrary machines (nodes) in a network. Hence the left branch is designed with this assumption in mind. Persistent entity beans are designed to be distributed each one on a different machine if this is desired [Tate+2003]. The programmer of an EJB need // should not know (at least if it comes to the intention of the initial EJB designers) where her beans are intended to be executed.

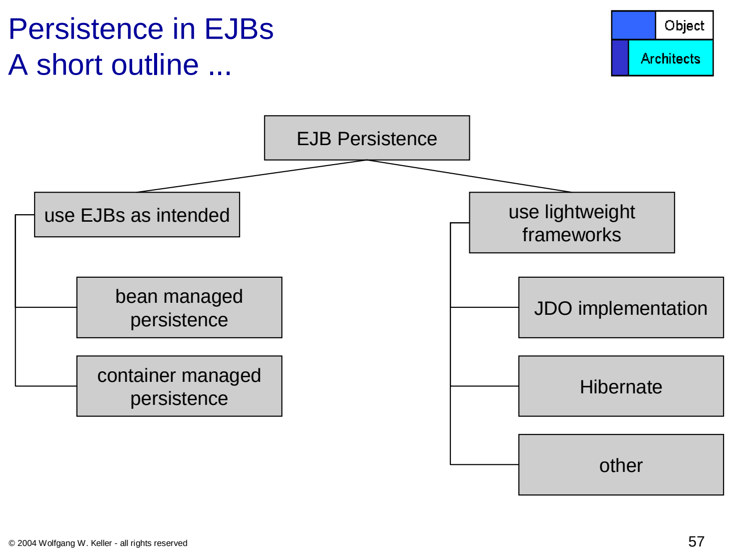


Figure 14: Persistence Options for EJBs

Lightweight Design

In practice the above assumption is somewhat too heavy. This is why people use other solutions if they know something about the execution environment. A normal execution environment would be an application server with all the functionality needed for the core of one application on one application server. If this doesn't provide enough throughput you may add more identical servers plus a load balancer plus some fail over capability.

If you follow such an application design (right branch), you are able to use only stateless session beans together with a lighter persistence framework for your business logic. Here the Java community discusses JDO [JoRu2003, Roos2002] versus e.g. Hibernate [BaKi2004]. And this is where we stop here because you can work with both – only the level of elegance and ease is being discussed.

Intended EJB Design

If you stick to the initial EJB design and assumptions you would write your business logic as EJBs (left branch). You then have the choice of Container Managed Persistence (CMP) versus Bean Managed Persistence (BMP).

BMP is basically a roll your own persistence manager. We have seen above that implementing all the features of a persistence manager will result in some effort. You should better delegate this to an application container vendor – unless you have a very special application with very special performance requirements. Many developers tend to over-estimate the need for individuality. CMP should really be an exception. If you want an idea of what the code looks like, look at [Sun2001].

CMP defines an interface for automatic persistence. The specification does not prescribe how this has to be implemented. So you better look twice what you container vendor provides. Prominent container vendors will provide you with tool sets to map your stuff to widespread relational databases (like DB2, Oracle, MySQL, and SQLServer to name a few). Other vendors might even use some for of flat or indexed files or an object database. The great advantage of CMP is that you get a Query Language at object level for free. One of the major disadvantages is platform dependency and also the need of very talkative deployment descriptors.

Section Summary

You might leave this section with a lot of open questions and an "IDontKnowWhatIShouldDoNow" Feeling. This is tough to prevent. The forces and considerations for EJB persistence fill roughly a third of Bitter EJB [Tate+2003] – a full size book. What this chapter could do was to make you 80 percent buzzword proof plus point you to a book, which you should not miss if you intend to use some form of persistent EJBs in your project.

8. A Few Remarks on Persistence in .NET

Persistence solutions in .NET used to be somewhat behind what you could find in the Java, Smalltalk and C++ arenas. This is somewhat strange as VB.NET is by far object oriented enough to allow implementing all the features explained above. Therefore we will just mention it and will not try to explain it.

The analogous tool to JDBC in .NET is ADO.NET (see e.g. [Hami2003]). But as we have seen in the code samples above the "quasi embedded SQL" side of it makes only the lowest layer of an object/relational access layer for relational database.

Today we see more players from the former C++, then Java and now C# arena jumping especially on the C#.NET wagon as the great similarity with Java makes porting a straight activity. Once you have a C# port a VB port is not far away.

Just to name a few players that have shown up on the screen we mention:

- **Pragmatier:** for C#.NET and VB.NET: They are between the first few who used to have a solution for VB for some years. See http://www.pragmatier.com/ .

- **FastObjects:** (formerly known as POET) who have made an evolution from a C++ OODBMS vendor to a Java Access Layer vendor to a .NET persistence layer provider. You could call that experience. See http://www.fastobjects.com/ .

- **db4o:** An object database with open source origins which can be used at no charge for private use. See http://www.db4o.com/ .

- **objectspaces:** and Microsoft is working on their own framework for .NET under the name objectspaces. They have their own newsgroup news://microsoft.public.objectspaces

The above list is far from complete but should give you an impression that .NET is catching up seriously and that it should be no problem to find the features mentioned in the above chapters also in .NET solutions.

Section Summary

Using Microsoft products is no longer an excuse for using some “handcrafted” persistence solutions. Products are beginning to evolve and what you find is in most cases way cheaper than handcrafted solutions which start simple and can quickly bring you deep into effort creek[7].

9. Summary

Know Your Application Style: This article has shown you that mapping objects to a relational database is by far not the only viable solution to persist objects. There are some application scenarios (single user, small amount of data, check in // check out) where lighter solutions like stream persistence may do an even better job.

Know the Concept of Transparent Persistence: We did not explicitly show you but you will have seen it from the code examples that what you get in persistence layers and interfaces is by far not transparent, fully orthogonal persistence. Why then mention the concept. Because it’s always good to know the maximum solution if you judge available solutions.

Don’t Develop Your Own Persistence Layer: Implementing a first CRUD pattern may be fun. But section 6 should have given you an idea of what kinds of features you

[7] which is a version of yoghurt creek which is a clean word for another creek where you don’t want to stay if you want to keep your job

need to implement for a full blown persistence layer. And be warned – the section is NOT complete. Experts will have mentioned that we left away such common things as n:m relationships, a lot of the mapping story and aspects such as performance optimization, prefetching and the like. Sooner or later you will need all these features in a major project and each of them means effort – and lots of them mean effort creek.

Know Some of What is Going on Under the Hood: If you follow the advice and buy a product you will need to judge it before you buy and you will need to have an idea of what goes wrong if something goes wrong e.g. with performance. And Murphy says it will. The literature cited here will give you hints if it happens.

10. References

[Atk+1989] **Malcolm P. Atkinson, François Bancilhon, David J. DeWitt, Klaus R. Dittrich, David Maier, Stanley B. Zdonik:** *The Object-Oriented Database System Manifesto*. in *"Deductive and Object-Oriented Databases"*, Proceedings of the First International Conference on Deductive and Object-Oriented Databases (DOOD'89), pp. 223-240

[BaKi2004] **Christian Bauer, Gavin King:** *Hibernate in Action*, Manning Publications, to appear in 2004. You can have a view on the current state by following the public review process at http://theserverside.com/resources/HibernateReview.jsp

[Chau2003] **Akmal Chaudhri:** *Java and Databases: The Good, the Bad and the Ugly;* Talk at JAOO 2003. Slides available from the organizers via http://www.jaoo.dk/

[Col1998] **Jens Coldewey:** *Choosing a Database Technology*, Tutorial at Comdex/Object World 98, Frankfurt, Germany. Available from http://www.coldewey.com/publikationen/database.html#DBTechTutorial

[Dat2003] **C. J. Date:** *An Introduction to Database Systems,* 8th Edition, Pearson Addison Wesley (July 22, 2003).

[Fow2002] **Martin Fowler et al.:** *Patterns of Enterprise Application Architecture*; Addison-Wesley Professional Series, 2002.

[GOF1995] **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides**: *Design Patterns, Elements of Reusable Object-oriented Software*, Addison-Wesley 1995.

[Hami2003] **Bill Hamilton:** *ADO.NET Cookbook*; O'Reilly & Associates, 2003.

[JoRu2003] **David Jordan, Craig Russel:** *Java Data Objects*, O'Reilly & Associates, 2003.

[Kel1997] **Wolfgang Keller**: *Mapping Objects to Tables: A Pattern Language*, in „Proceedings of the 1997 European Pattern Languages of Programming Conference, Irrsee, Germany, Siemens Technical Report 120/SW1/FB 1997. Also available as an electronic document from the authors website http://www.objectarchitects.de/

[Kel+1997] **Wolfgang Keller, Jens Coldewey**: *Relational Database Access Layers: A Pattern Language*, in „Collected Papers from the PLoP'96 and EuroPLoP'96 Conferences„, Washington University, Department of Computer Science, Technical Report WUCS 97-07, February 1997.

[Kel+1998a] **Wolfgang Keller, Jens Coldewey**: *Accessing Relational Databases: A Pattern Language*, in Robert Martin, Dirk Riehle, Frank Buschmann (Eds.): Pattern Languages of Program Design 3, Addison-Wesley 1998.

[Kel1998b] **Wolfgang Keller:** *Object Relational Access Layers: A Roadmap, Missing Links and more Patterns*, Proceedings EuroPLoP Irrsee, Germany, UVK Universitätsverlag Konstanz 1998. Also available as an electronic document from the authors website http://www.objectarchitects.de/

[ODMG2000] **Rick G. G. Cattell (Ed.) et. al.**: *The Object Data Standard 3.0;* Morgan Kaufmann Publishers, 1993.

[Pol+2001] **Macario Polo et al.:** *Reflective CRUD.* Proceedings EuroPLoP 2001, Irrsee, Germany, UVK Universitätsverlag Konstanz 2001. Also available from the authors' web site at http://www.inf-cr.uclm.es/www/mpolo/yet/ . Link checked 2003-12-30.

[Roos2002] **Robin M. Roos:** *Java Data Objects*; Addison-Wesley Professional Series, 2002.

[Sun2001] J2EE Documentation: Commented example of a CMP bank account Entity bean at http://www.objectarchitects.de/ObjectArchitects/orpatterns/EJBPersistence/AcountEJB.htm

[Tate+2003] **Bruce Tate et al.:** *Bitter EJB*, Manning Publications 2003.

[Thor2001] **Graham Thornton:** *Optimistic Locking with Concurrency in Oracle*; can be found as PDF at http://www.orafaq.com/papers/locking.pdf . Link checked 2003-12-30.

[TOP2003] **Oracle Corp:** TOPLink User Manual.