A Critical Analysis of JDO in the Context of J2EE

Axel Korthaus, Matthias Merz
Department of Information Systems
University of Mannheim
Schloss, D-68131 Mannheim, Germany
{korthaus|merz}@wifo3.uni-mannheim.de

Abstract

The Java Data Objects (JDO) industry standard appears to provide a promising framework for persisting Java objects in an efficient way. Many experts even regard JDO as a more appropriate approach to persistence management in J2EE-based enterprise application environments than the built-in Entity Bean components, which are an integrated part of the J2EE set of APIs. In this paper, we present a critical analysis of JDO in the context of J2EE and compare it with Entity Beans using bean-managed and container-managed persistence. The analysis is based on literature research, practical experience, and the results of performance measurements in an exemplary experimental setup.

Keywords: Java Data Objects, Enterprise JavaBeans, Entity Beans, persistence, performance

1. Introduction

One of the many challenges of developing distributed multi-tiered enterprise application systems is how to provide an appropriate architecture for persisting business data. Numerous requirements have to be met, e.g. with regard to performance, concurrency, transactions, interoperability, maintainability etc., and any of today's technologies have their drawbacks in one or the other respect. While object-oriented databases (OODBs) appear to be the conceptually most appealing kind of data store for the backend, relational databases (RDBs) are still prevalent, and for simple purposes even file system-based solutions are used. Therefore, developers often have to fight the so-called impedance mismatch between the object-oriented and component-based business logic application layer on the one hand and the conceptually different data stores on the other hand. Thus, for separation of concerns, the data stores have to be encapsulated by an object-oriented abstraction layer that can be used seamlessly by the object-oriented application code.

In the Java world, the widespread Enterprise JavaBeans (EJB) component model [9] as part of the J2EE platform [12] represents a popular server component approach to the development of distributed client-server enterprise applications. It offers infrastructure services such as persistence management in an "out-of-the-box" fashion via EJB component containers. Persistence is only one of those services, among transactions, security, activation/passivation etc. Data that has to be persistent is normally modeled with Entity Beans which can provide bean-managed or container-managed persistence (BMP vs. CMP). Although Enterprise JavaBeans as a whole seem to be indispensable in the Java world as a fully-fledged server component technology, the new Java Data Objects (JDO) standard [11] might well be a considerable alternative to Entity Beans as the persistence framework of choice.

In this paper, we will describe these technologies, i.e., JDO, BMP, and CMP, in the context of J2EE, discuss their pros and cons, and present results from our practical experience with exemplary applications, especially focusing on performance. We have implemented a simple library application in two different variants. The first variant uses a JDO-based persistence approach with a session bean and plain Java classes, while the second variant relies on CMP-based entity beans representing the data to be persisted. The results of our performance measurements which were performed on an Intel Pentium IV 1,6 GHz machine with 512 MB RAM, Windows 2000, a MySQL 3.23.53 database using InnoDB tables, the application server JBoss 3.2.0 [1], and Signsoft's JDO implementation intelliBO 3.1 [7] round off the findings. We will have a look at the question whether the JDO-based approach, the EJB 2.0 CMP-based approach, or the BMP-based approach implemented with JDBC is most beneficial with regard to performance and good design, respectively.

2. Enterprise JavaBeans (EJB)

Enterprise JavaBeans (EJB) is the server-side component architecture for the J2EE platform. It enables the development of distributed, transactional, secure and portable Java enterprise applications. One of its most beneficial features is that it frees the developer from having to deal with code that handles transactional behavior, security, connection pooling, or threading, because these tasks are delegated to the application server and

EJB container providers. Thus, the developer can fully concentrate on programming only the business logic, while the option to provide the “plumbing” code manually still remains.

Among the different types of enterprise beans defined by the standard, namely stateless session beans, stateful session beans, entity beans, and message-driven beans, only entity beans are designated for persistence.

Entity beans can be used to represent persistent business objects both in software applications and their corresponding design models. As opposed to normal Java objects which do not include transactional persistence management, entity beans encapsulate persistence mechanisms, hiding them from any calling code. Thus, the container is enabled to optimize persistence features while keeping the data store open and flexible, only to be determined at deployment time [8]. So, entity beans represent persistent data that can be shared across multiple simultaneous remote and local clients.

We already mentioned the two ways of providing persistence with entity beans, namely bean-managed persistence (BMP) and container-managed persistence (CMP). With BMP, the entity bean developer is responsible for writing code to save the bean’s state. Using CMP, on the other hand, the full power of the EJB architecture can be utilized, because no persistence code has to be written by the developer, only some configuration settings regarding the container-managed fields have to be made in the bean’s deployment descriptor. Furthermore, since it can be programmed against the container API, the bean implementation class is independent of the concrete data store at hand.

Because of the flawed EJB version 1.1 standard, entity beans were very much under discussion, since they had serious performance and other issues. Since version 2.0, many of the counter-arguments have become invalid. The specification has been improved considerably, especially because of the introduction of local interfaces and CMP 2.0. Entity Beans using CMP 2.0 have to be implemented using an abstract bean class with abstract methods representing the persistent fields of the entity bean and by declaring an abstract persistence schema. Through this approach, the deployment tool or the container, respectively, are able to generate the concrete class at deployment time and to optimize persistence management automatically.

The general process of getting an Enterprise Bean running, no matter whether it is an entity bean or not, is quite complex and comprises a number of steps. It is generally divided into a development phase, on the one hand, which is independent of a concrete application server product and is guided by the standardized EJB specification, and a deployment phase, on the other hand, which also involves product-specific activities. In detail, these steps are:

- developing the enterprise bean source code including all relevant classes and interfaces such as the bean class, local and remote component and home interfaces;
- preparing the deployment descriptor (`ejb-jar.xml`) which provides a declarative description of the enterprise bean and is evaluated by the EJB container;
- compiling the Java source codes;
- building the `jar`-file of the enterprise bean which is an archive containing the enterprise bean `class`-files and the deployment descriptor;
- deploying the `jar`-file in an EJB container in a product-specific way. Usually this step comprises the specification of a product-specific deployment configuration file, e.g. `jboss.xml`, and the introduction of the `jar`-file(s) containing the new enterprise bean to the container.

3. Java Data Objects (JDO)

With the specification of Java Data Objects (JDO), a new standard has been established which provides for the transparent storage of Java objects. JDO provides standardized access to persistent objects and, in addition, enables the use of various data store types from different vendors (e.g., relational databases, file systems, object-oriented databases). JDO almost completely handles the transparent persistence of application objects. There is no need to explicitly manage the storage and retrieval of individual fields of the Java objects to be persisted [13]. JDO distinguishes between persistence-capable, persistence-aware and transient classes [11]. Persistence-capable classes are those whose instances are allowed to be stored in a JDO-managed data store. Persistence-aware classes are those that while not necessarily persistence-capable themselves, contain references to managed fields of classes that are persistence-capable. Thus, persistence-capable classes may also be persistence-aware.

The persistence needs of Java classes using JDO have to be described in an XML-based persistence descriptor. Usually, the “ordinary” classes are then compiled, and afterwards, the code to handle persistence is added to the compiled `class` files in a separate step called JDO enhancement. Hereby, the byte-code enhancer modifies the class file by providing methods that implement the `PersistenceCapable` interface, which represents the primary JDO hook for persistence management, thus facilitating field access by the JDO infrastructure. This process of byte-code enhancement has the big advantage that it produces full binary compatibility of the enhanced classes with respect to different JDO implementations. In the ODMG standard, which was a main influence factor for the JDO specification, class enhancement is vendor-specific, so there is no application binary portability across implementations.

However, the approach as such has been criticized by many developers who do not trust in an enhancer tool making changes to their `class` files which they cannot see and check. However, the standard does not prescribe byte-code enhancement, it is also possible that the class developer explicitly implements the `PersistenceCapable` interface in his source code.

Besides the general feature of transparent persistence as described before, JDO also provides persistence by reachability, also called transitive persistence, which is defined, according to [11], as follows: "When a persistent instance is committed to the data store, instances referenced by persistent fields of the flushed instance become persistent. This behavior propagates to all instances in the closure of instances through persistent fields." Other important features provided by JDO are change tracking and automatic enlistment, persistence by inheritance, lazy data loading, and automatic navigation of the object graph [13].

An application that needs to read or write persistent objects first has to lookup or otherwise obtain a `PersistenceManagerFactory` and use an instance of this factory to obtain a `PersistenceManager` instance. Then, methods such as `makePersistent()` to store a new object or `getExtent()` to read lists of objects can be invoked on the `PersistenceManager`.

JDO includes a sophisticated identity mechanism to be able to extend object identities to the data store. The standard distinguishes between an application identity ("primary key" concept), where values in an instance determine its identity, a data store identity which is independent of any instance values and a non-durable JDO identity which guarantees uniqueness in the Java Virtual Machine but is not supported in the data store.

Similarly to the EJB standard, the JDO standard includes its own data store-independent query language, the JDO Query Language (JDOQL), which can be used to write queries based on conventional Java operators, so the developer does not have to use SQL or worry about table structures. As with EJB, the developer is not forced to write persistence infrastructure code or map objects and attributes to tables and columns by himself. And since it is a standardized API, JDO-based applications are independent of specific JDO product vendors and work with any compliant JDO implementation as is the case for EJB-based applications with regard to EJB-compliant container and server products.

Since we are focused on the application of JDO in the J2EE environment, it should be mentioned that JDO usage is envisaged in two environments, namely non-managed, i.e., non-application server environments, and managed environments like J2EE, where the Java Connector Architecture [10] has to be used to specify the contract between the JDO vendor and an application server. JDO's potential for use within J2EE applications will be discussed in the following section.

4. Comparison and discussion of BMP, CMP, and JDO in the J2EE context

Keeping the premise that we need to develop a distributed enterprise application with server-side components in Java, we have several more or less attractive options to get persistence in the J2EE context. One option is to use entity beans with CMP, another is to use entity beans with BMP (and JDBC, or JDO, for example), and a third option is to do without entity beans completely, using a stateless session bean façade [4] together with JDO or JDBC. In the following subsections, we will briefly analyze these alternatives.

4.1. CMP entity bean vs. BMP entity bean + JDO/JDBC

Since EJB 2.0 introduced essential performance improvements and vital features such as container-managed relationships, it is broadly agreed upon that using entity beans with CMP is normally the faster and better approach than using entity beans with BMP. It is obvious that CMP eases the developer's job by requiring a lot less coding, but at first, it might be surprising that CMP is supposed to be faster, because one would perhaps expect an additional overhead for container-managed services. However, as Roman et al. explain ([6], p. 366): "CMP entity beans, if tuned properly, are much higher performing than BMP entity beans. For example, with BMP, it takes two SQL statements to load an entity bean: the first to call a finder method (loading only the primary key) and the second during `ejbLoad()` to load the actual bean data. A collection of `n` bean-managed persistent entity beans requires `n+1` database calls to load that data (one finder to find a collection of primary keys, and then `n` loads). With CMP, the container can reduce the `n+1` database calls problem to a single call, by performing one giant `SELECT` statement. You typically set this up using container-specific flags (which do not affect bean portability)."

According to [8], CMP enables many optimization possibilities within the container and the container-generated database access code. For example, unnecessary expensive database calls can be avoided, because the container can monitor the in-memory buffer of a bean and can therefore avoid storing the buffer to the database before committing a transaction if the buffer has not changed. These are important reasons why CMP solutions are generally faster than BMP solutions.

[3] adds to the reasons for using CMP over BMP. For example, EJB containers can support pessimistic and even very powerful optimistic in-server locking models a BMP entity bean cannot take advantage of. If a respective CMP engine is used, the developer does not have to care for multi-request transactional issues, collision handling, etc. Moreover, BMP entity beans have to provide finder and select methods which are manually implemented using complex JDBC or SQL code and cannot profit from the EJB Query Language (EJB-QL) features which are used to configure queries without having to put additional code into the bean class. Another argument against using BMP is that container-managed relationships (CMRs) cannot be used and complex and error-prone code has to be written to implement the relationships. Furthermore, BMP-based solutions cannot do cascading deletes and cannot create database tables or generate primary keys in a portable way.

However, there are some scenarios where BMP for use in an entity bean can be appropriate or might even be required. As Jewell [3] states, this is always the case if you try to accomplish something that cannot be done through CMP, as for example,

- *interaction with stored procedures:* CMP engines normally don't provide features to interact with stored procedures in database management systems;
- *use of nonstandard SQL*: if the SQL submitted to a database has to be customized with proprietary extensions to use vendor-specific features, this might not necessarily be supported by a specific CMP engine;
- *collecting data from multiple stores*: CMP engines usually do not support entity beans pulling data from multiple databases, which is uncommon but might be necessary in a legacy system integration context.

Furthermore, as Tulachan [14] remarks, if you are writing persistence logic to a very proprietary legacy backend system that you are intimately familiar with, or if your transaction and security requirements are very fine-grained, or if you want complete control of managing persistence, then you have to choose BMP. Jewell [3], however, points out that "if you choose to use a BMP entity EJB when a CMP engine or a stateless session EJB with JDBC could be used, you're taking an unnecessary performance hit."

4.2. CMP entity bean vs. stateless session bean + JDO/JDBC

The approach of employing a stateless session bean as a session bean façade [4] which uses JDO or JDBC to directly interact with the data store has become quite popular, especially because of the flaws of older EJB specifications. Jewell [2] argues that "EJBs have transactional, security, and interoperability support at the container level. It's been repeatedly demonstrated, however, that these capabilities are more relevant at the SLSB [stateless session bean] facade layer that accesses a data layer." So, in many cases it can be sufficient to have a session bean interact with the client and use JDBC or JDO as a more lightweight approach to persistence in the background.

If you consider the effort of developing and packaging entity beans (cf. section 2) even if the entity bean is used to represent just pure persistent domain objects with no business logic involved, this seems like an "overkill" [13] or a "sledgehammer to crack a nut" [5]. Gopalan postulates that simple data objects should not be represented by entity beans, and claims: "It is objects that contain business logic and processes that have to be componentized to maximize code reuse." [13]

Now, using JDBC or an O/R mapping tool to implement the persistence layer has again numerous drawbacks [5]. JDBC coding is complex, not really object-oriented, SQL must be used, and so you get tied to relational data stores. The use of O/R mapping tools often locks the developer into a particular vendor, foreclosing application portability. So, JDO appears to be the best technological approach to be combined with stateless session beans in a J2EE environment, and also provides several advantages over entity beans, as will be discussed in the following subsection.

4.3. Strengths and deficiencies of JDO

The main problem with using entity beans for persistence, as McCammon [5] puts it, is that "with Entity Beans you end up polluting your coarse-grained component model with fine-grained objects from your domain model, all because you wanted persistence." If the benefits provided by the EJB standard can be realized on the session bean layer, all that is still needed is to persist graphs of pure Java objects in a standard way without having to deal with a database directly. Using JDO, the heavyweight entity bean approach to getting persistence, involving the packaging up of home and local interfaces, a bean class and a deployment descriptor and deploying this component can be avoided. Session beans as a felicitous concept can be leveraged for component-based development, and JDO can be used as the lightweight framework to persist the domain model objects representing the business object layer. Since JDO is an industry standard, database and vendor independence can be achieved. Compared to entity beans, it is much easier to develop with JDO, because you work with normal Java classes representing your domain model in a one-to-one mapping as if you would write any normal Java program, but getting persistence as a side-effect. A big advantage over entity beans is that you need not work with your deployment tools and application server right from the beginning, but can build, run, and debug your domain model implementation as a normal Java program and, only after testing outside the application server environment, add the component interfaces in a layered fashion [5]. It is also criticized that entity beans do not support inheritance, EJB containers do not support multithreading within entity beans, and the use of OODBs with CMP entity beans is not possible at the moment, whereas JDO does not have these problems. JDO is scaling quite well, and objects are loaded only if they are needed or if the programmer explicitly demands it.

However, there are also a number of deficiencies of JDO. As Gopalan [13] states, the JDO Query Language, which is totally different from EJBQL, requires additional parsing processes during query execution, thus making it more resource intensive than existing object-oriented query languages. The use of String filters in JDOQL is another problem, because potential typos in the query strings can only be detected at runtime. A good alternative query language is S.O.D.A. [15], which in our opinion should be used as paragon for the improvement of JDOQL.

As already mentioned, many people have strong objections to byte code enhancement. One serious problem is for example, that if the developer uses subclasses of `Collection`, he has to make sure that a JDO implementation is used which is able to support these classes during enhancement. Although there are implementations that support all relevant subclasses, such as intelliBO and Lido, this is not prescribed by the JDO standard 1.0. As has been proved, the undesirable and relatively expensive enhancement process can be circumvented, since a good alternative would be to use the Java reflection API to get meta information about the class structure.

Perhaps one of the most severe problems with the current JDO standard is related with the concept of relationships. At the moment, JDO 1.0 does not offer any managed (inverse) relationship support. However, it has been announced that this problem will be solved in one of the upcoming versions of the JDO standard. Nevertheless, several JDO product vendors already have integrated proprietary solutions to the problem of bi-directional relationships.

4.4. Performance of basic CRUD operations with CMP, BMP, and JDO

To get a first impression of basic performance behavior of the different approaches, we measured CRUD (create, read, update, and delete) operations on a platform as described in section 1. We had a session bean that either used JDO directly or accessed an entity bean with either BMP (using JDBC) or CMP to create, read, update, and delete 5.000 instances of a business object. The results of the measurements are shown in table 1. As can be seen, for this rather contrived test scenario with its basic operations and standard tool configurations, CMP cannot show its strengths, and JDO is able to keep pace. However, these results should not be generalized as we will explain in more detail in the context of the library example application described in the next section.

Table 1: Performance measurement (CRUD with 5.000 objects with CMP, BMP and JDO)

Implementation	Create	Read	Update	Delete
JDO: average [sec]	24,5	22,5	25,9	25,3
variance[sec^2]	13,4	8,4	4,7	14,5
CMP: average [sec]	29,9	30,8	34,0	26,9
variance[sec^2]	3,9	0,1	0,1	0,0
BMP: average [sec]	29,7	27,6	23,4	23,3
variance[sec^2]	3,9	33,7	0,0	0,0

5. The library application example

In order to get a better practical feeling of the use of JDO in a J2EE environment and to be able to compare a JDO-centric design with a CMP-centric design based on a concrete implementation, we implemented a simple library application example. Main components of this problem domain are the books in the library, library members who borrow books, and library accounts of members which are used to register loans. Each library member is associated with exactly one library account, and each library account can list any number of loan events. A loan comprises the set of books a library member borrows at a specific point in time. Figure 1, which shows the JDO version of our application design, gives an overview of these problem domain classes, modeled with UML. `BookDO` represents books in the library, described by only a few typical attributes. Each library member (represented by `LibraryMember`) has an `id`, a `name`, an `address`, and an association with an account object (of class `LibraryAccount`). The account object has an attribute `accountLock` which indicates whether the library member is currently authorized to borrow books or not. If a library member borrows one or more books, a new loan object (of class `Loan`) is created and provided with the current date and the references to the borrowed books.

The experimental setup and the products that were used have already been mentioned in section 1. We have implemented two variants of this application, one using JDO together with a stateless session bean, and one using CMP-based entity beans. However, in both cases we have used the same backend, i.e., a MySQL 3.23.53 database, and the same JDBC driver both for the CMP engine in

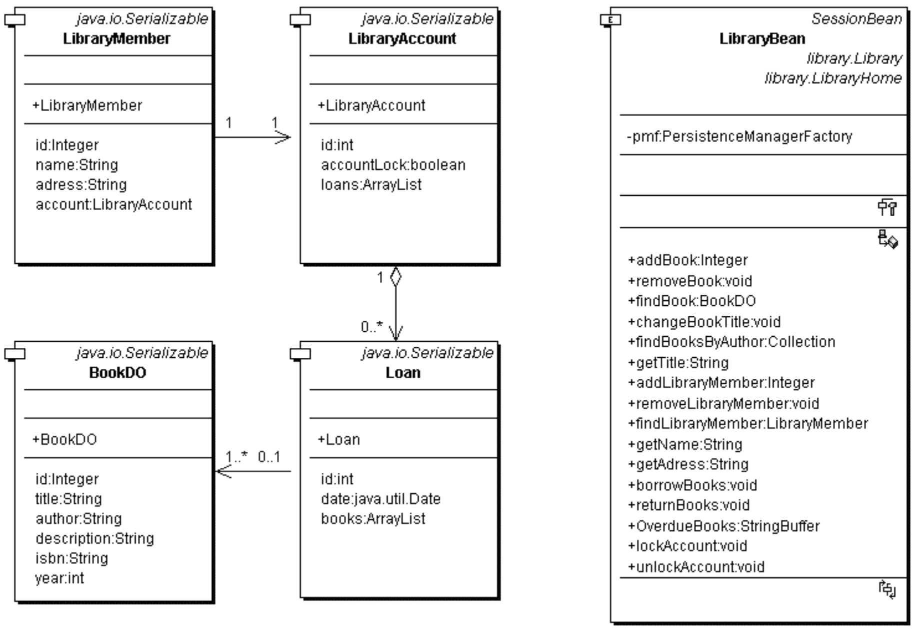

Figure 1: The JDO-based variant of the library application

JBoss and for the JDO implementation from intelliBO (namely, MySQL Connector/J 3.0.6). Furthermore, similar container configuration settings with respect to transaction management, connection pooling etc. have been used in both cases.

The architecture of the example application uses well-known J2EE design patterns (cf. [4]). For example, in both cases a session bean façade (`LibraryBean`) encapsulates the business and persistence logic and represents the interface to the client, which is the same for both implementations. The session bean façade is a basic design pattern used to minimize network calls if multiple entity beans are involved, to provide a more efficient transaction management for large transactions, and to encapsulate business processes. One more design pattern has been applied in both variants to reduce the amount of network calls, namely the Data Transfer Object pattern. Class `BookDO` is used as a data transfer object to encapsulate all relevant attributes of a book, so that they can be transferred within one single network call between the client and the session bean, thus avoiding multiple calls for accessing single fields.

In the CMP-based variant, all the business objects (book, loan, library member, library account) are represented by entity beans. To avoid performance penalties, the session bean façade accesses these entity beans via local interfaces only, which is only possible since EJB version 2.0. In the JDO-based variant, plain Java objects were used to represent the business objects. These objects had to be enhanced, and for the intelliBO JDO implementation to be able to work within JBoss, the vendor-specific JCA implementation (included in file `ibo.rar`) had to be plugged in.

Now, the following test scenario was applied for both implementation variants: first, a test client randomly creates 20.000 books and 5.000 library members to populate the data stores. Then, 300 library members each borrow one book, 200 members each borrow two books, and 100 members three books each. The loan date is always set to a value less than four weeks, which represents the normal loan period. In a second iteration, additional loan events are generated with loan dates exceeding the allowed period, so that the borrowed books are overdue: 100 x 1 book, 200 x 2 books, and 300 x 3 books. In total, this sums up to 1200 borrowers and 2400 borrowed books.

For our performance measurement, we decided to select a complex business method that involves not only the retrieval of instances of one single class, but also requires navigation across associations. The `OverdueBooks` method in the session bean was a suitable candidate for that goal. Its responsibility is to retrieve a list (of type `StringBuffer`) of the ids of all library members who have currently borrowed at least one book, together with the ids of those borrowed books that are overdue, ordered by borrower.

The `OverdueBooks` method was implemented as follows. First, a query (a JDO query in the case of JDO, and a `findAll` method in the case of CMP) retrieves a collection of all member ids, then the corresponding account is fetched (which involves container-managed relationships in the case of CMP), and, if existing, loan objects for the account are retrieved. If at least one loan object exists, the member id is put into the result `StringBuffer` object and the loan date is checked. If the loan period has been exceeded, the corresponding books are fetched and their ids are added to the result object.

We measured the execution time of the `Overdue-Books` methods for the two different implementations several times, and the analysis resulted in an average execution time of 61,3 sec for the JDO-based solution (variance 11,8 sec^2) and 17,4 sec for the CMP-based solution (variance 0,2 sec^2). This result shows, that CMP is performing a lot better in this specific setting than the stateless session bean plus JDO-based solution, which supports the positive assessment of CMP many authors share. However, a totally different result was achieved when the `OverdueBooks` method was called twice without restarting JBoss in between: while the CMP-based solution consumed the same amount of execution time, the JDO-based solution had an enormous speedup and now only needed 6 sec which must be due to intelligent caching. It is obvious that factually no realistic general statements about performance can be made, because performance depends on numerous factors, among which are different tool configuration settings, different tool features, different usage patterns etc.

6. Conclusion

In this paper, we presented a detailed analysis of different approaches to getting persistence in a J2EE-based environment. As we have pointed out, the best solution with regard to performance and simplicity usually will be to choose entity beans (version > 2.0) with CMP or a session bean façade approach combined with JDO. Generally, the developer should decide between these two options. Exceptional cases where other solutions, especially BMP-based approaches, can be more beneficial have been mentioned in the paper.

The various advantages of a well-configured CMP 2.x-solution together with container-managed relationships have been mentioned in detail. If bi-directional relationships can be dispensed with, or it is acceptable to fall back on a proprietary solution, JDO will be a good choice because of its lightweight nature. Moreover, the use of JDO is especially worth while if frequent accesses to the same objects occur, because the built-in caching mechanism can then speed up the program execution time. Beyond the basic caching features, most of the existing JDO products can be tuned further, e.g. with respect to query performance optimization, either on the SQL level by providing additional hints, or by optimizing query results and query fetch-groups.

To conclude with, during our theoretical and practical work with JDO we got the impression that the JDO standard is very promising and will certainly play its role in the (enterprise) Java world.

References

[1] JBoss (2003): http://www.jboss.org/

[2] Jewell, T. (2002): The Subtlety and Power of Entity EJBs. In: WebLogic Developer's Journal Magazine, vol. 01, issue 03, http://www.WebLogicDevelopersJournal.com

[3] Jewell, T. (2002): Why Choose a CMP Architecture? In: WebLogic Developer's Journal Magazine, vol. 01, issue 04, http://www.WebLogicDevelopersJournal.com

[4] Marinescu, F. (2002): EJB Design Patterns. Advanced Patterns, Processes, and Idioms. Wiley Computer Publishing, New York etc.

[5] McCammon, K. (2003): JDO & J2EE. Commentary. http://www.jdocentral.com/JDO_Commentary_Keiron McCammon_1.html

[6] Roman, E., Ambler, S., Jewell, T. (2002): Mastering Enterprise JavaBeans. 2nd ed. Wiley Computer Publishing, New York etc.

[7] Signsoft (2003): http://www.signsoft.com/de/intellibo/

[8] Sucharitakul, A. (2001): Seven Rules for Optimizing Entity Beans. http://developer.java.sun.com/developer/technicalArticles/ebeans/sevenrules/

[9] Sun Microsystems (2002): Enterprise JavaBeans Specification. Version 2.1. Proposed Final Draft http://java.sun.com/products/ejb/docs.html

[10] Sun Microsystems (2002): J2EE Connector Architecture Specification. Version 1.5. Proposed Final Draft 2. http://java.sun.com/j2ee/connector/download.html

[11] Sun Microsystems (2002): JSR-000012 Java Data Objects (JDO) Specification (Final Release), version 1.0, http://java.sun.com/products/jdo/

[12] Sun Microsystems (2003): Java 2 Platform Enterprise Edition 1.4. Proposed Final Draft. http://java.sun.com/j2ee/j2ee-1_4-pfd3-spec.pdf

[13] Suresh Raj, G. (2003): Java Data Objects. http://my.execpc.com/~gopalan/java/jdo/jdo.html

[14] Tulachan, P. (2002): JavaLive! session on Practical Enterprise JavaBeans. Chat. http://developer.java.sun.com/developer/community/chat/JavaLive/2002/jl0305.html

[15] SourceForge S.O.D.A. - Simple Object Database Access: http://sodaquery.sourceforge.net/