Integrating the Java 2 Enterprise Environment with Java Data Objects

Stefan Böttcher
University of Paderborn
Fürstenallee 11 ,
D-33102 Paderborn , Germany
email: stb@uni-paderborn.de

Robert Hoeppe
Poet Software GmbH
Kattjahren 4-8,
D-22359 Hamburg, Germany
email: robert@hoeppe.de

Abstract. Whenever object-oriented applications that use the Enterprise JavaBeans (EJB) environment have to store data in a database, there is a need to combine EJBs with access to persistent data. The current use of entity beans for persistent data storage has a number of weaknesses. Component level inheritance and dynamic querying are not supported appropriately, and the whole approach seems to be overweight. In order to overcome these weaknesses, we implemented a JCA[1]-compliant resource adapter, which enables the use of Java Data Objects (JDO) instead of Entity Beans and allows the introduction of a lightweight, easy to use and powerful transparent persistence layer for the Java 2 Enterprise Environment (J2EE).

1. Introduction

1.1. Problem Origin and Motivation

There is a growing interest in using Enterprise JavaBeans (EJB) for distributed, object-oriented applications (e.g. [28]). An EJB-based application server provides important features like scalability, security, connection and transaction management, reuse of application components and transparent persistence via container managed persistence entity beans. While these features are needed in our E-business application, there are additional key requirements, namely to provide a natural integration of object-oriented data storage and dynamic queries. The aim of our contribution is to enhance EJB by both a natural support of object-oriented domain modeling and efficient dynamic querying possibilities.

This work has been carried out at Poet Software GmbH, where the principal purpose was to combine the features of the EJB component model with the advantages of an OODBMS in order to develop a way to enable flexible query facilities in hierarchical

[1] J2EE Connector Architecture [21].

and object-oriented data structures, such as hierarchically organized product catalogs. We demonstrate this by means of a subsequent example in Section 2.1.

1.2. Relation to other Work and our Focus

As mentioned, Enterprise JavaBeans are becoming more and more popular as a middleware platform. According to [5,15], common middleware services like EJB must be able to guarantee quality of service-functionality within the context of dynamically changing environments and requirements. Another contribution [5] also combines elements of component middleware and model-integrated computing however, it approach combines the OMG's model driven architecture [13] with CORBA instead of with EJB.

We follow the argumentation of [28], that fine-grained entity bean (EB) design leads to higher communication efforts, while a coarse-grained design usually ends up with the need for a self-controlled dependent object structure. Several contributions [1,6,10,11,15] offer various design patterns to overcome this weakness. We adopt the ideas of [1,6,11] to use the so called Session Façade Pattern, but our approach uses Java data objects (JDOs) instead of EBs.

Another contribution [6] proposes a solution different from EBs and introduces a persistence layer implementation which is extended in [9]. Furthermore, [9] draws up a list of requirements for an ideal persistence layer and concludes that JDO meets nearly all of them. Different from these contributions, our approaches supports dynamic querying, inheritance, and a natural integration into J2EE, just to name a few.

The work of [29] discusses the integration of EJB with, among other things, Java Blend [22] and JDBC [24]. However, our approach integrates EJB with JDO, that can be regarded as the successor of Java Blend. Our approach is further supported by a comparison of JDO and JDBC in an unmanaged environment [26], that states that JDO allows for a reduction in the lines of code and for easier development and testing cycles. These results are transferable to the modeling of bean managed persistence entity beans in a managed environment, however in our contribution, we suggest to replace the entity beans with JDOs.

Finally, a contribution [17] suggests the integration of JDO into the J2EE environment, but does not substitute EBs with JDOs. In contrast, we follow [11] to use JDOs instead of EBs. In addition, we present an implementation architecture for the integration of EJB with JDO, using an OODBMS. This approach efficiently supports dynamic queries and inheritance that respects the capability of continuous object-oriented modeling.

2. Problem Description

2.1. Application Requirements

Our example application operates in a large, hierarchically organized product catalog. As such, it must be expected to satisfy the following requirements:

- The type of underlying storage system might change in the future or could even be a combination of different storage types (for example, XML data, OODBMS or ERP/legacy systems). Thus there should be a backend-independent data interface that offers transparent access to each possible data store. Ideally, there should be no impedance mismatch between the object-oriented domain model of the application and the data storage.
- The number of client requests and the computational effort of business methods typically increase over time. Hence, the application must be highly scalable.
- It should be possible to include third party components and to reuse given functionality in different contexts or interrelations.
- The application requires that data store systems support transactions.

Application servers along with container managed persistence solve some of these requirements, such as scalability and service abstraction, but not all of them. For instance, they cannot provide a flexible mechanism for retrieving data (queries) that can be modified according to differentially compounding attributes and relationships for each request (as discussed in Section 3.3.3.). A simple example of such a scenario might look like this:

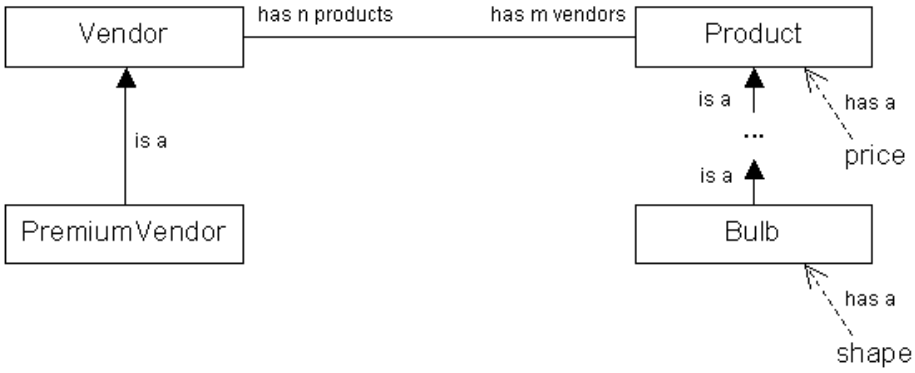


Fig. 1. Simplified example of a hierarchically build-up object model

This example contains a simple n:m relationship and some instances of single inheritance. A potential query on this structure could possibly be:

"Which PremiumVendors can provide candle-shaped light bulbs priced under 50 cents?"

The requirement is to answer dynamic queries similar to this one efficiently in a Java 2 Enterprise Edition Version 1.3 compliant application server.

2.2. Restrictions of the Current EJB Concept

Let us take a closer look at the EJB 2.0 [20] component model to get a grasp of its possibilities and restrictions with respect to the above-stated requirements.

2.2.1 The EJB Component Model

The following figure shows the principal constituents of an entity bean (EB) in its interaction with a client.

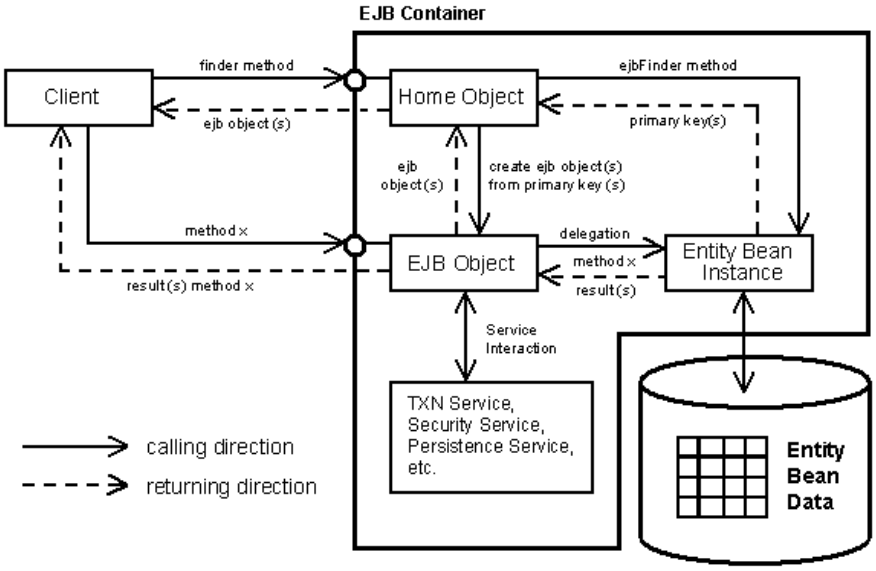


Fig. 2 Exemplification of the EJB 2.0 component model

An Enterprise JavaBean is a deployable component controlled by an application server's container. An EB is a persistent, distributed object and consists among other things of a Home Object, an EJB Object, and an entity bean instance. As an entity bean is also a component, the client will never call any of its methods directly, only indirectly through the EJB object, which itself manages the service interactions and delegates the call to the associated entity bean instance.

There are two possible ways to persist an EB: either using container managed persistence (CMP) or bean managed persistence (BMP). Using CMP, the persistent

object description is declared in the bean's deployment descriptor. The container then generates the backend-specific code at deployment time. Using BMP, persistence management code (for example, using JDBC) has to be written in the bean manually.

2.2.2 Runtime Queries

In order to find an existing EB in storage, a client has to use the predefined finder-methods of the bean's Home Object, which delegates the client's request to the query's implementation in the entity bean instance. According to the EJB-QL query language (which was newly introduced in [20]), a query might be parameterized, but the query itself remains static, because the bean instance is generated at deployment time, not at application execution time (runtime). Using BMP, it is possible either to parse the finder method's parameter or to pass a JDBC query string that can be executed inside the data store, though both approaches have restrictions (as discussed in Section 3.3.3.).

2.2.3 Inheritance

After the successful execution of a finder method, the client holds references to one or more EJB Object instances of the entity bean it was looking for. Suppose the result is a collection of Vendor instances from the example above and the client wants to compare them with instances of the Vendor's subclass PremiumVendor. The object-oriented way of programming would be to use the equals() function of the class. (The equals() method would internally cast the Vendor instance to a PremiumVendor. The following code shows this explicitly for clarity.)

```
Boolean result = premiumVendor.equals((PremiumVendor)vendor)
```

However, this will not work for enterprise beans, because the client is not holding a true reference to a distributed object but to a distributed object handle, that is, to the EJB Object. We must distinguish two cases. In the first case, the client resides outside the application server, thus accessing the EJB object via its remote interface. In this scenario, the client handle is a stub. In the second case, the client is accessing the EJB object from inside of the application server. But even when using local interfaces (which are newly introduced in [20]), the client handle is a true EJB Object reference, not an enterprise bean instance reference. Although it is possible to simulate inheritance by delegation, true component level inheritance is not addressed by the EJB specification [7,20,25]. A more common solution of the second case is described in Section 3.

2.3. The Aim of our Contribution

We integrate EJB naturally with a data storage access technique that supports each of the following:

- dynamic queries

- transparent persistence [16]
- continuous object-oriented modelling (including inheritance).

3. Our Contribution to Solve the Problem

3.1. The Application Programmer's View: Substitute Entity Beans with Java Data Objects

Our approach to the problem is to replace the entity bean with a persistence capable, plain Java object complying to the Java data objects (JDO) specification [23]. That is, to exchange the EJB persistence mechanism with the one provided by Java data objects, as described in the following subsections. In order to access these objects remotely, the widely accepted Session Façade Pattern [1,6,11] can be applied.

3.1.1 JDO: A Short Introduction

The Java data objects specification is an industry standard for object persistence [16,23]. It provides a unified, simple and transparent persistence interface between Java application objects and transactional data stores, that makes it a valuable persistence layer for transactional middleware environments. The JDO specification consists of a lightweight API framework allowing developers to easily persist complex object domain models without the need to care about specific mappings for data storage.

3.1.2 The Example in JDO

Since Java data objects (JDOs) are just plain Java objects, inheritance is naturally supported. With the JDO-QL query language, dynamic querying is also possible. Using the example from Section 2.1, the query filter is

```
premiumVendor.products.contains(product)
&& product.price < 50
&& ((bulb)product).shape.startsWith("candle")
```

In conclusion, the integration of JDO into EJB supports each of our stated requirements: dynamic queries, transparent persistence and continuous object-oriented modeling (including for instance inheritance).

3.2. The System's View: Embedding JDO in the J2EE Architecture

In addition to other areas of application, JDO is appropriate as a persistence layer for transactional middleware environments like the Java 2 enterprise architecture. Though it is possible to use the JDO API in an enterprise bean in the same way one would in a stand-alone application in an unmanaged environment, there would be no benefit

from the managed environment contracts (described in detail in the J2EE connector architecture (JCA) specification [21].)

3.2.1 The J2EE Connector Architecture

The J2EE connector architecture [21] provides a standard architecture for connecting the J2EE platform to heterogeneous enterprise information systems (EIS). It is composed of a framework of several client- and server-side interfaces and defines a standard set of system-level contracts [19,21] between an application server and an EIS. A so-called resource adapter implements the EIS-side of these contracts.

3.2.2 The JCA Management Contracts

The following figure provides an insight into our implementation of a JDO native resource adapter with local transaction management. The main system-level contract interactions are outlined by the relations between the different interfaces.

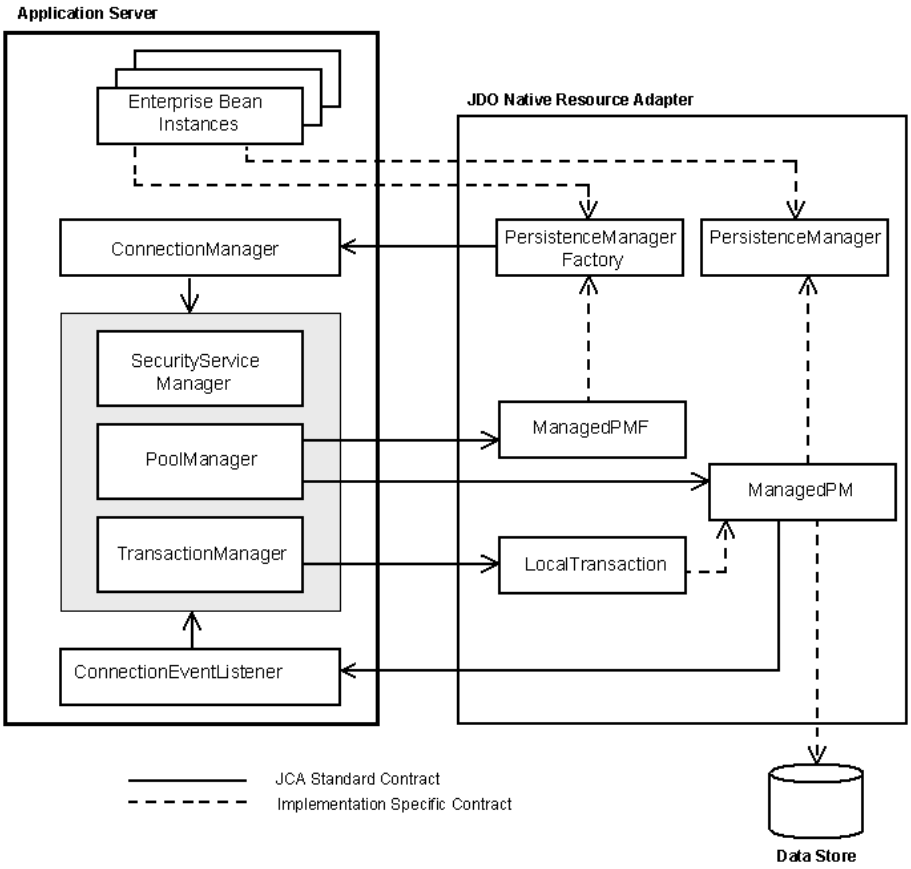


Fig. 3 Application scenario outlining the JCA management contracts

The most important aspects of the integration are the transaction, security and connection management contracts. They enable an enterprise bean to benefit from the application server's system services using the usual JDO client interfaces such as the PersistenceManager [23] and its factory. For example, the transaction management contract enables the application server to take over transaction control from the JDO implementation, while the resource adapter provides the necessary synchronization with the JDO implementation's data store. Furthermore, user-coded, bean-managed, container-synchronized transaction demarcation is possible.

3.3. Alternative Implementations for Transparent Persistence in EJBs

With a JDO application server integration based on the J2EE connector architecture, there are different use case scenarios for applying JDO to enterprise beans. These are discussed in the following subsections, where our contribution is compared to existing solutions for transparent persistence.

3.3.1 Our Favorite Approach: Using Façade Pattern with JDO

We propose to model the domain object model with JDOs and make them accessible by remote clients by encapsulating them with stateless session beans. This approach is well known under the term Session Façade Pattern [1,6,11] and is actually used to encapsulate entity beans in a similar way. The pattern is demonstrated using the following subscription (see the figure below). A circle in the Session Bean Façade box represents a session bean, a circle in the JDO Domain Model box represents a Java Data Object, and an arrow represents a potential method call.

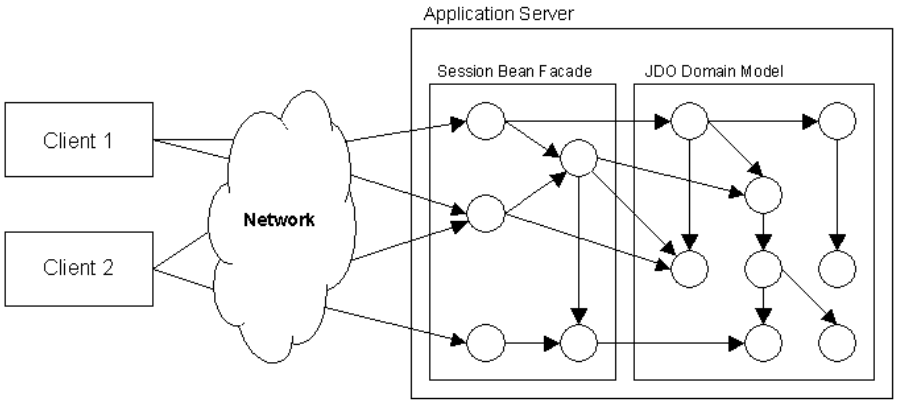


Fig. 4 Architecture of the Session Façade Pattern in combination with JDO

Using this design pattern, we rely on a widely accepted and mature development model, which leaves us with two alternatives: either to model the business logic of our application in the session bean layer or within the Java data objects. Using the first alternative, we could benefit from the advanced pooling and caching mechanisms which application servers offer to session beans. Stateful session beans could also be

used alongside coupled client-server interaction. For example, a bean could hold the object ID of a JDO which represents a client's user data object, for instance a shopping cart. Using the second alternative, JDO provides even further advantages, such as the easier development cycle or the opportunity to use the object model also in an unmanaged environment (which will be described in detail in Section 3.4.)

For the realization of an asynchronous and more reliable interaction between client and application server, the client can operate on the newly introduced Message Driven Beans (MDBs) [15,20] instead of session beans to access the domain object model. This pattern is called Message Façade Pattern [11], and describes the encapsulation of the domain model with MDBs. Similarly to the Session Façade approach, we propose to use Java data objects instead of entity beans to build up the domain model.

3.3.2 Other Approaches Using JDO

As proposed in [17], the JDO API could also be used for entity beans with bean managed persistence (BMP). This would provide a standard data store-independent way to implement the EB's persistence mechanism and would avoid the restrictions entailed with BMP EBs (concerning data store-independent dynamic queries.) However, inheritance in EBs still remains unsupported using this method.

JDO could alternatively be used in container managed persistence (CMP) for implementing container-generated persistence functionality. Thus, it is possible to hide a complex fine-grained domain object model behind a coarse-grained entity bean façade, and to plug in different JDO implementations for CMP in the application server. However, this approach is neither capable of handling inheritance directly nor of overcoming the restriction of dynamic querying CMP EBs. Furthermore, its implementation remains application server specific.

3.3.3 Non-JDO Approaches

As stated in 2.2.1, dynamic querying remains possible using bean managed persistence entity beans, but in contrast to our proposal this has the following disadvantages. On the one hand, query expressions, for instance encoded in JDBC, are not data store-independent. On the other hand, queries that compute a join on dynamically given attributes between two or more data sets are in principle possible, because the BMP code, CMP code or even the client could locally compute the join from these data sets. However, this would ignore the query optimization already implemented in the data store and result in large amounts of data traffic and consequently inferior performance compared to efficient query processing operations inside of the data store. In addition, neither of the specified scenarios supports inheritance.

Some proprietary solutions like WebGain's TopLink [27] or Castor JDO [4] provide similar persistence layer functionality to JDO, for example, dynamic querying and inheritance. However, these approaches are unlikely to become an agreed industry standard, thus locking the user into non-portable, one-vendor solutions. Furthermore,

most non-JDO approaches focus on relational data stores accessed via JDBC, or rely on a proprietary data storage providing even less data store independency. In almost every case, the application server integration is proprietary as well, while a JCA-compliant JDO resource adapter is executable in every J2EE 1.3-compliant application server. Therefore, our beans and JDOs remain portable in every J2EE-compliant environment.

3.4. Integrating Existing J2EE-Applications with JDO

As Java data objects are just plain Java objects and can be accessed via a standardized Java API, all parts of application code of a local J2EE application – including all kinds of beans and even servlets or JSP code – can benefit from the use of them. Using the two Façade Pattern presented in Section 3.3.1., even non-local application parts may access them. Likewise, a Java data object is free to interact with any kind of Java objects inside the scope of the application server – or even beyond it, as it may use any kinds of enterprise beans, too. Thus it appears that the proposed integration of Java data objects is fully interoperable with the Java 2 enterprise environment.

3.5. Further Advantages of our Contribution

In addition to the previously mentioned abilities of our approach, there are further advantages worth considering. By using JDO, we enable continuous object-oriented modelling from the application to the data store without impedance mismatch, because the application uses JDOs that can be directly mapped to an OODBMS (in our implementation to Poet's OODBMS "FastObjects" [14]). Note that this not only avoids the impedance mismatch for the application programmer, but also for the data mapping from the JDOs to the data store. The more an object domain model relies on inheritance and inter-object linking, the more important are the advantages of an OODBMS [2,12]. Beyond the hierarchically build-up product catalog example above, these advantages apply, in particular, to CAD/CAM, medical technology and geospatial applications.

Another advantage of JDOs over entity beans concerns the complexity of the development cycle. With JDO, we basically start with a Java object model which is testable without using any persistence functionality at all. The business logic is then tested using the JDO API functionality. Finally, the session facade is introduced and the JDOs are deployed along with the session beans in the container. In contrast, an EB can only be tested inside its operational environment, namely the application server's container. This alone increases the amount of work in deployment and in more complicated debugging processes, besides the overhead needed for coding a component compared to coding a Java data object (which is a plain Java class).

Further, a JDO domain model (as composed just of plain Java objects) is not restricted to an application server's realm, but can also be executed and persisted in any unmanaged environment supported by JDO implementations, even down to handheld

devices. Therefore, by using this approach and with respect to the Java virtual machine, the JDO object model can be shared between completely different target platforms.

Finally, according to [16], our approach will have additional benefits in adaptability and maintainability that may improve the development time (and therefore cost) of enterprise applications by 20%, with some analysts suggesting that the actual savings may be much higher.

4. Summary and Conclusions

Our implementation integrates the following advantages into a single approach: transparent object persistence, portability of application code, vendor- and backend-independence, dynamic query capability, and object-oriented design capabilities for both business objects and the object domain model down to the data store level without impedance mismatch or restricted inheritance support. It also includes all known advantages of the J2EE middleware architecture like scalability, transaction and security services, and the component model approach, just to name a few and is furthermore fully interoperable with all parts the architecture.

Today, even architectural IDEs that support the model driven architecture [5,13], e.g. Interactive Objects' ArcStyler [8], cannot solve the intrinsic weakness of the EJB architecture concerning dynamic queries. Although they are capable of automatically generating finder methods for an EB's primary key attributes corresponding to predefined CMP relationships, it is not appropriate to generate finder methods for every single bean attribute or even for each possible combination of bean attributes occurring in dynamic queries. However, with our approach, these tools can generate code that offers transparent object persistence via Java data objects instead of entity beans, yielding all the advantages mentioned.

In a more sophisticated application environment, an application server might have to deal with distributed transactions across multiple application servers and heterogeneous data storages. The J2EE Connector Architecture addresses this scenario, enabling resource adapters to take part in distributed transactions [19,21]. Therefore, our method of using JDO makes an application totally unaware of even a composed underlying data storage.

Finally, we are not restricted to use Java data objects merely in combination with enterprise beans. It seems promising that the integration of JDO with Java Servlets and Java Server Pages may open a wide range of new application areas that is, whenever the need for lightweight application servers (like Apache) in addition with object persistence arises, the use of Java data objects will be appropriate [16,17].

References:

1. Alur, D.; Crupi, J.; Malks, D.: core J2EE Patterns. Sun Microsystems Press, Prentis Hall PTR, New York (2001)
2. Atkinson, M.P.; Bailey, P.J.; Chisolm, K.J.; Cockshott, W.P.; Morrison, R.: An approach to persistent programming. Computer Journal 26 (4) (1983) 360-365
3. Banerjee, A.; Harkey, D.: Persistance Mechanisms in the Java 2 Platform, Enterprise Edition (J2EE): Designing for Performance. JavaOne, San Francisco (2002)
4. Castor JDO. http://castor.exolab.org/jdo.html
5. Gokhale, A.; Schmidt, D. C.; Natarajan, B.; Wang, N.: Applying Model-Integrated Computing to Component Middleware and Enterprise Applications. Submitted for review at ACM CACM 02 (2002) http://www.isis.vanderbilt.edu/~gokhale/PDF/CACM02.pdf
6. Grand, M.: Java Enterprise Design Patterns, Patterns in Java, Volume 3. John Wiley & Sons, New York (2002)
7. Heik, S.: Introduction to Java Data Objects (JDO). (2001) http://www.inqmy.com/
8. Interactive Objects: ArcStyler. http://www.io-software.com
9. Kruszelnicki, J.: Persist data with Java Data Objects, Part 1- 2. JavaWorld (2002) http://www.javaworld.com/javaworld/jw-03-2002/jw-0301-jdo.html
10. Koch, V.: Objektpersistenz für EJB-basierte Applikationserver. Component Developer's & User's Forum, Frankfurt (2001)
11. Marinescu, F.: EJB Design Patterns. John Wiley & Sons, New York (2002)
12. Morrison, R.; Connor, R.C.H.; Cutts, Q.I.; Kirby, G.N.C.: Persistent Possibilities for Software Environments. In: The intersection between Databases and Software Engineering. IEEE Computer Society Press (1994) 78-87
13. Object Management Group: Model Driven Architecture (MDA). OMG Document ormsc/01-07-01 edition (2001) http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01.pdf
14. Poet: FastObjects. http://www.fastobjects.com/
15. Roman, E.; Ambler, S. W.; Jewell, T.: Mastering Enterprise JavaBeans Second Edition. John Wiley & Sons, New York (2002)
16. Roos, R.: Java Data Objects. Addison-Wesley (2002)
17. Russell, C.: Using the Java Data Objects Specification With the Enterprise JavaBeans (EJB) Specification. JavaOne, San Francisco (2002)
18. Schantz, R. E.; Schmidt, D. C.: Middleware for Distributed Systems. Chapter in The Encyclopedia of Software Engineering. John Wiley & Sons (2001)
19. Sharman, R.; Stearns, B.; Ng, T.: J2EE Connector Architecture and Enterprise Application Integration. Addision-Wesley, Boston (2001)
20. Sun Microsystems: Enterprise JavaBeans Specification, Version 2.0. Palo Alto (2001)
21. Sun Microsystems: J2EE Connector Architecture Specification, Version 1.0. Palo Alto (2001)
22. Sun Microsystems: Java Blend, Version 2.0. http://wwws.sun.com/software/javablend/index.html

23. Sun Microsystems: Java Data Objects Specification, Version 1.0. Palo Alto (2002)
24. Sun Microsystems: JDBC Specification, Version 3.0. Palo Alto (2001)
25. Uhlig, M.: Evaluation von Java Data Objects. Großer Beleg, TU Dresden (2002)
26. Van Echelpoel, K.: Java Data Objects: A Revolution in Java Databanking? Universiteit Antwerpen (2002) http://www.vanechelpoel.be/thesis/pdf/KrisThesisENG.pdf
27. WebGain: TopLink. http://www.webgain.com/products/toplink/index.html
28. White, J.: Enterprise JavaBean architecture and design issues. SIGSOFT, IEEE Computer Society, Washington, DC (2001)
29. White, S.; Cattell, R.; Finkelstein, S.: Enterprise Java Platform Data Access. SIGMOD 98, Seattle (1998)