MIND THE GAP! - MAPPING ENTERPRISE JAVA OBJECTS TO THE DATABASE

David Parsons – Valtech

Summary

One of the key challenges facing an enterprise Java system is the integration of object-oriented systems with corporate data in relational databases, not only to map data structures but also to support generic services such as security and transactions. This paper reviews the main technology options: JDBC, EJB, JDO and object-relational mapping tools. In particular it focuses on the relative merits of EJB container managed persistence against direct mapping with a JDO tool. Discussion will centre on which strategies are appropriate in different e-Business scenarios. Oracle 9iAS will be used as the example platform, with TopLink as the mapping tool.

> '*One of the secrets of success for mapping objects to relational databases is to understand both paradigms, and their differences, and then make intelligent tradeoffs based on that knowledge'* - Scott Ambler [Ambler 2000]

Objects and the database

Relational databases are the predominant storage technology in the current IT market and thus provide a common component of software systems. They are frequently a part of 'legacy' systems with which new projects must integrate. They are well understood and their underlying table model, which uses repeated data keys to relate normalised tables to each other, is based on sound mathematical foundations. Of course not all types of application with persistent data should necessarily be based on relational data. Applications such as CAD, CASE tools or other check in / check out types of applications, where transactions are very long and there are no collisions should use other forms of storage [Keller 1997] but in general, RDBMS are an effective solution to storage requirements and will therefore be a challenge frequently encountered by the object-oriented developer. In joining together an object model with a database schema, there are three basic scenarios:

1. *Deriving the database schema from the object model.*
   This is possible where no legacy database exists, or any existing schema can be revised at will. Although this is the ideal scenario from an object-oriented perspective, it is not very common.
2. *Deriving the object model from the database schema.*
   This can be done where a complete database schema is already in place for the system to be developed, but is unlikely to result in an acceptable object model. At the very least, we might expect a level of denormalisation to take place between the relational schema and the object design.
3. *'Meet in the middle' mapping*
   This is where both a database schema and an object model exist and they need to be mapped to one another. This is the most likely scenario and the one that is most likely to result in both an acceptable database schema and an acceptable object

model, (though there may well have to be some compromise in either or both of these).

Given the likelihood of having to support 'meet in the middle' mapping, developers need techniques and tools to join together two very different models, the object-oriented design and the relational schema. Since the beginnings of object-oriented programming, this has been a recurrent problem in making objects persistent, since relational database schemas do not map easily to an object-oriented view of a business model. Trying to bridge this gap without either understanding the issues or using appropriate tools can lead to unsatisfactory design and poor performance.

The Object-Relational Impedance Mismatch

At first glance there seems to be a simple relationship between objects and rows in a database table. An object is defined by a class, which defines (among other things) the structure of its internal data. Similarly, a database table has a schema that defines its columns. An instance of a class (an object) contains data in the same way that a row (or tuple) contains data (Figure 1).

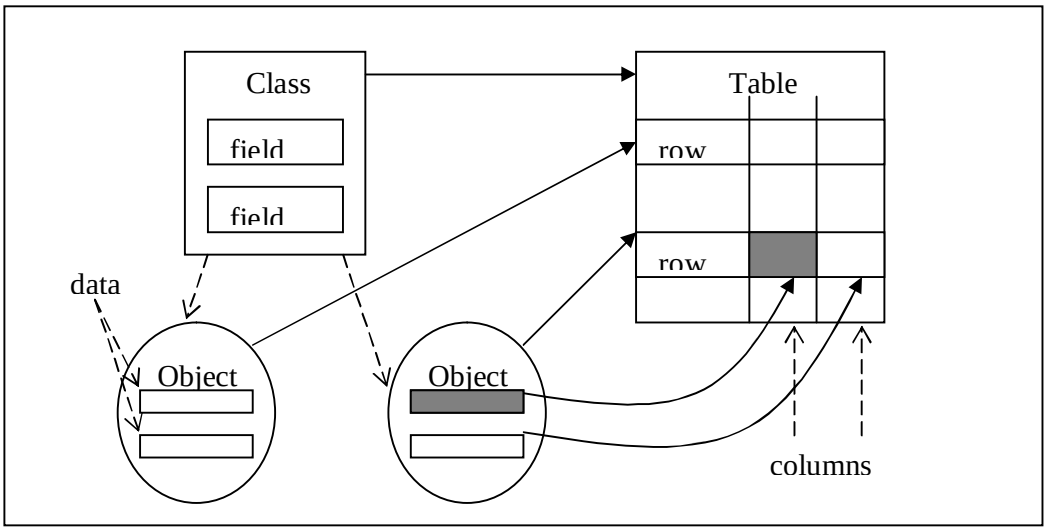

Figure 1 – the relationship between classes and tables, objects and columns, fields and rows

However, although there are some apparent similarities between an object-oriented model and a database schema, there are many areas where there is not a particularly good fit between the two technologies. This is often known as the 'impedance mismatch'. This mismatch covers a number of problem areas:

- Unique object and table identifiers
- Mapping of data types
- Relationships and normalization
- Inheritance
- Operations

Unique object and table identifiers:
Java objects in a virtual machine are identified by references that in effect represent the memory space that the object occupies at a particular time. Two objects can have identical state, but be differentiated by their memory location. Data in a database, however, must have unique key data. Therefore any attempt to persist objects in a database requires that they have unique object identifiers (OIDs), effectively key fields, and a suitable mechanism for generating these unique keys, typically integer based but in some cases including type identification. It should be noted that these keys should not have any relationship to the actual business data in the objects or tables, and need to be generated in such way that an appropriate level of uniqueness is achieved without excessive locks on the database [Ambler 2000]. Utilising the database to generate keys is probably the most effective solution, though it is proprietary and therefore not portable. Many other strategies exist, however, and have been documented elsewhere [Marinescu 2002]. It is up to the developer to select an appropriate strategy for the particular context in which they are working and to ensure that any persistable object is provided with a suitably unique and immutable key at the point of creation. Although such OID key fields are orthogonal to the object domain model, they could also be used in the object code in some circumstances to reduce memory consumption [Brown & Whitenack 1996].

Mapping of data types:
Although simple data types have straightforward conversion mappings between Java and the database (e.g. String to VARCHAR and vice versa), more complex types do not have a natural mapping. For example, Boolean types in Java often have to be mapped to character or number based fields, and serialized Java objects are frequently stored as BLOBS or other binary representations such as LONG RAW, effectively losing their semantics. It is also sometimes necessary to use finer grained components in the database than those in the object model, so that one field of an object may have to be mapped to more than one column of a database table.

Relationships and Normalization:
Objects that might be regarded as a single instance in an object model might be seen very differently in a normalised database schema. Whereas an object might maintain relationships using collections of objects, a database will need to use many tables and keys. What appears as a simple many to many relationship in an object model requires complex foreign key management in the database. There are different strategies to consider too, for example modelling one to many relationships can be done using either back pointers or a separate table. Alternatively a one to many relationship represented in an object model by a collection of references may simply be written to a single field as a serialized object. In all of these scenarios, the implementation of the object model must take account of the actual data representation in the database and handle type conversions and table navigation appropriately.

Inheritance:
Relational databases do not directly support inheritance, so objects that partake in an inheritance hierarchy have to be modelled indirectly in the database. There are a number of strategies to do this, each with their own advantages and disadvantages, but a common solution is to join multiple tables with common keys to represent the classes in the hierarchy. Although a reasonable design choice this can lead to poor performance, requiring other less elegant solutions [Brown & Whitenack 1996].

Operations:
Objects are inherently encapsulations of both process and data, with operations that relate specifically to the data in the object. In contrast, database operations such as stored procedures are not tied to particular object representations, so only the data from an object can be directly stored in a database, not its other characteristics. This usually means that objects can be aware of their data representation in the database but not vice versa: the object-relational metadata is not in the database but implicitly in the code or explicitly in code related descriptors such as XML files. This means that forward engineering a database schema from an object model is likely to be more successful than reverse engineering an object model from a database schema, since the reverse engineering provides no operational semantics for the objects thus derived.

Technologies for Object-Relational mapping

There are a number of technologies for O/R mapping, which are more or less transparent depending on the level of tool support. We can categorise these into two approaches: custom written persistence using JDBC on an ad-hoc, per-application basis, or transparent persistence using some mapping payer between client code and the persistence layer. The implementation of such a persistence layer can be based on proprietary development, a third party tool and/or recognised standards supported by multiple vendors.

JDBC:
JDBC is an API for Java that provides a SQL call level interface to an RDBMS. Although this is perfectly adequate for simple database access, particularly where most access is read only, it is not the best solution for more complex systems. Because all that it allows us to do is to execute SQL against the database and receive rows in result sets, we have to make our own semantic connections between our object model and this data. For example, to persist a 'Module' object to the database using JDBC we might set up a Java PreparedStatement coding the basic framework of a SQL INSERT statement:

```
PreparedStatement writeModule = connection.prepareStatement
 ("INSERT INTO MODULE VALUES (?,?,?,?,?)");
```

Into this PreparedStatement we would then need to transfer data from an object, one field at a time, for example (using an object called module1):

```
writeModule.setInt(1, module1.getModuleID());
```

Then, finally, the update statement can be executed:

```
writeModule.executeUpdate();
```

Even for trivial mappings between objects and the database, writing JDBC is a laborious task, even more so where mappings are complex. It is likely that hand coding the persistence layer to translate between objects and relational data will take up to 30 or 40% of the project development time [WebGain 2001], and is not a simple task. The developer has to overcome all the problems of impedance mismatch and ensure

that the resulting system is performant and transactionally robust. In addition, the direct exposure of SQL statements in Java source code means that the Java application layer is tightly coupled to the particular SQL syntax set of the database that it is accessing. Any changes to the database platform propagate directly into the Java code, requiring maintenance and recompilation. Many project teams have tried and failed to produce an industrial strength persistence layer using a call level interface such as JDBC.

Transparent persistence:
This is any technology that hides the call level interface behind a higher level API. There are many technologies that provide for transparent persistence and provide a range of services. In particular, we expect a reduction in the amount of code required to manage the persistent objects [Barry & Stanienda 1998], and in many cases an improvement in performance due to features such as caching. Transparent persistence mechanisms come in four general forms:

1. Custom written persistence layers, typically based on wrapping JDBC
2. Proprietary persistence tools
3. Entity Enterprise Java Bean (EJB) container manager persistence tools
4. Java Data Objects (JDO) compliant persistence tools

These vary in a number of ways, for example in simplicity of use, level of intrusion into the business code and support for infrastructure services such as transactions.

The core services of a persistence layer can be summarised in a short list [Kruszelnicki 2002]:

- Create, read, update and delete operations on objects
- Support for transactions (start, commit, rollback)
- Management of object identity
- Caching
- Creating and executing queries

Custom written persistence layers:
Writing a persistence layer is a challenging task, though the JDBC API does provide the necessary support for queries, result sets, stored procedures and transactions. In addition, the JDBC 2.0 APIs include support for enterprise level database access including connection pools and data sources for scaleable distributed access. Custom written persistence layers can result in much greater reuse of persistence code across applications within an organisation, and are preferable to the constant rewriting of JDBC code for standard actions. The difficulties tend to arise where there is a great deal of write and update behaviour that requires sophisticated transaction management, and in maintaining performance for high demand systems. Transactions must be handled at the code level, and there is no intrinsic caching behaviour in the JDBC APIs. For areas that go beyond the basic JDBC APIs, such as distributed transactions, the persistence layer will have to call out to other services. Depending on the nature of the system, Keller and Coldeway estimate a required development effort of between .5 and 35 person years for a relational database access layer [Keller & Coldeway 1997]. Because of the complexity of an enterprise persistence layer, and the fact that any such effort is to a large extent reinventing the wheel, it is not a recommended strategy, and many developers have looked to third parties to provide persistence tools that can increase productivity, reliability and performance.

Proprietary persistence tools:
Since the beginnings of the widespread adoption of object-oriented technology in the later 1980s, there have been many tools developed to provide object persistence. These have been typically either object-oriented databases or object relational mapping tools, though mappings to many other forms of persistence, including flat files and legacy EIS, have been built. Early versions of these tools were typically for C++ or Smalltalk, but the rise of Java has seen this language become the focus of persistence tools in recent years. Tools that began life as Smalltalk mapping tools (such as TopLink) or C++ based object-oriented databases (such as POET) have been reapplied to the Java field. These tools generally provide sophisticated support for object persistence, including transaction management and, where appropriate, visual mapping editors. TopLink, for example has a visual mapping editor (the Mapping Workbench) that generates XML metadata to represent the relationships between objects and the database (Figure 2).

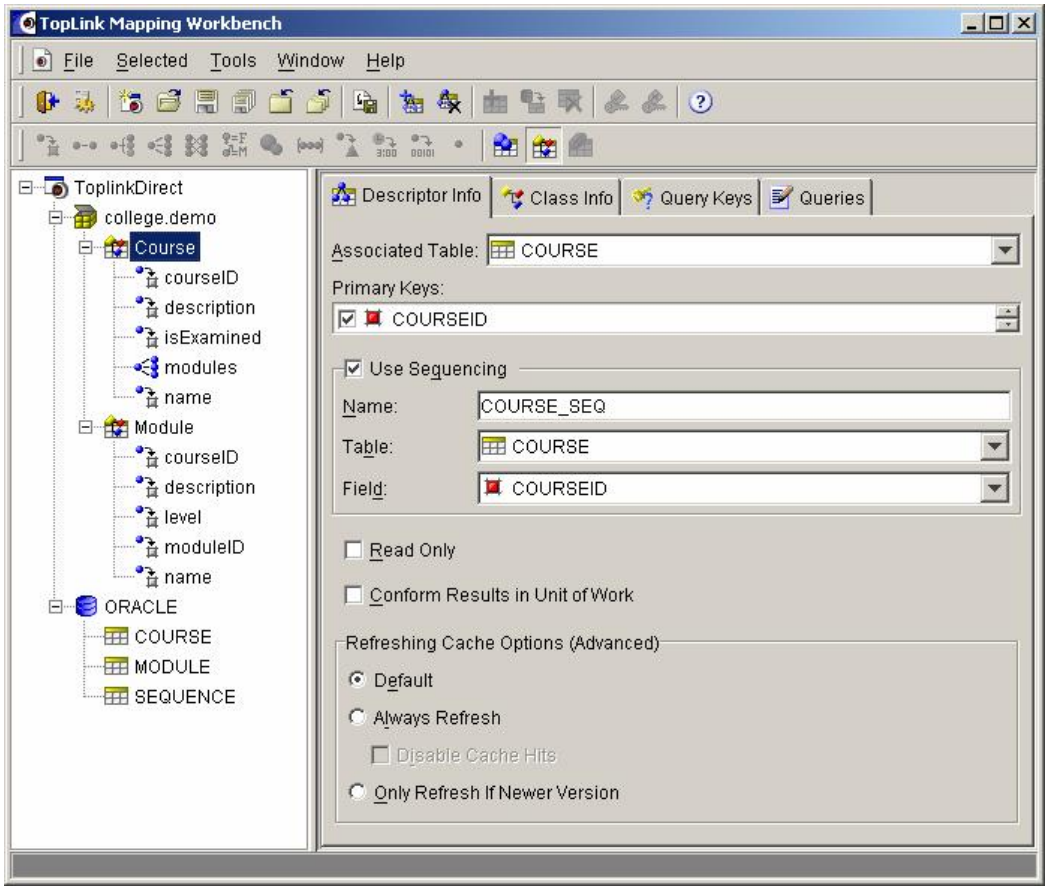

Figure 2: The Toplink Mapping Workbench

Using a mapping tool such as TopLink means that the Java layer of the system can be much more object-oriented than JDBC code, transparently rendering Java objects persistent with a few lines of code. For example, we can transactionally write an object to the database with a TopLink unit of work without needing to change the object or write SQL code

```
UnitOfWork unitOfWork;
unitOfWork = session.acquireUnitOfWork();
Course tempCourse = (Course)unitOfWork.registerObject(course1);
unitOfWork.commit();
```

Although these tools have been very productive in individual projects, their major drawback is their propriety nature. No two products work in the same way and therefore they all have the problem of vendor lock in. Because they generally involve some kind of intrusion into the code itself, even if this is just a few lines, they complicate any attempt to port an application to another platform or persistence mechanism. For this reason, more standardised forms of persistence such as entity beans and JDO have gained popularity.

Entity EJB container manager persistence:
The enterprise Java Beans specification [Sun 2001], currently at version 2.0, includes options for both custom written persistence using JDBC (or other APIs) and container manager persistence for entity beans. This specification includes object relational mapping, relationships, transactions and security. Much of the persistence configuration is expressed not in code but in XML metadata, which is used by the EJB container to generate appropriate persistence code. The main advantages of using this mechanism are that it is standard (so entity beans can be moved between different EJB containers) and that it enables the container vendor to provide persistence services such as caching and database failover, which are transparent to the application developer. Once a client has accessed an EJB then the code to manipulate the object is very similar to standard Java. Only acquiring references to objects is unlike standard Java, making use of remote stubs bound into JNDI and 'create' and 'find' methods based on the Factory Method pattern [Gamma et al 1995]

```
Context ctx = new InitialContext(env);
SequenceHome sequenceHome =
      (SequenceHome)ctx.lookup("Sequence");
Sequence sequence = sequenceHome.findByPrimaryKey
      (new SequencePK("COURSE_SEQ")));
```

However, although client code is relatively transparent the building of EJBs themselves is tightly coupled to the EJB framework, and requires adherence to a range of patterns and idioms that are unlike standard object development. In addition, EJB development means having to write as much information into XML deployment descriptors as Java code. Although there are usually tools to do this their functionality is by no means consistent across platforms.

There are also some drawbacks to this approach that mean it is not the holy grail of standardisation and portability. The specification does not cover all aspects of persistence, so some parts of the metadata are not standard. This complicates portability between different containers. Also, the current specification does not cover all possible types of mapping, so that vendors vary on the levels of mapping support that they provide with their container tools. Further, the EJB specification may be supported at different version levels by different application servers, so that we cannot, for example, port an EJB 2.0 application to a server that only supports the EJB 1.1 specification.

Java Data Objects (JDO):
JDO is a standard Java specification for a transparent persistence API, based partly on the Object Data Management Group's (ODMG) Java data binding specification, which is part of a wider multi-language specification that has been in development since 1993. JDO APIs are designed to make object persistence as non-intrusive as possible, so that the code of the objects themselves does not have to contain any persistence related syntax. In terms of client code, we might expect JDBC code to be three times as long as its JDO equivalent [Kruszelnicki 2002]. However, use of JDO is not entirely transparent, with specific operations having to be performed to explicitly persist and read objects to keep them in synch with the database.

```
JDOPersistenceManagerFactory factory =
      new JDOPersistenceManagerFactory(session);
PersistenceManager manager = factory.getPersistenceManager();
Course course1 = new Course();
course1.setxxx // etc..
manager.makePersistent(course1);
```

An important aspect of JDO is that is does not assume any particular type of data store is being used for object persistence. Whereas the vast majority of EJB persistence tools assume that entity beans will map to a relational database, JDO tools can map to different types of database or any other system that maintains data, indeed an application written using JDO could be transparently ported from one storage technology to another, or between Enterprise Information Systems, without affecting client code. It should be noted however that specific JDO implementations tend to map to one particular type of storage technology, so that one might use O/R mapping while another might use an ODBMS. Porting from one storage format to another would typically involve migrating to a different JDO implementation rather than reconfiguring the existing one. Also, although managed JDO implementations can theoretically wrap a J2EE connector architecture mapping to a range of back end system, the practicability of this depends entirely on specific implementations. The JDO specification is still in a relatively early stage of development, and leaves much unspecified, or makes general suggestions to vendors as to how certain features of the spec might be implemented. JDO works on two level, managed and unmanaged. Unmanaged JDO code is where the JDO implementation is working as a stand-alone tool to expose an API to client code, which provides persistence. A managed JDO implementation works within an application server container to integrate with its services such as security and transaction support. In a J2EE server, JDO is expected to integrate with the J2EE Connector Architecture (J2EE CA) to provide integrated persistence. This may be to database resources or to other systems that provide persistent data. A managed JDO system would provide transparent access to multiple types of data, giving a standard API for accessing multiple back end systems.

Selecting persistence strategies

The various options for object persistence each have their own advantages and disadvantages, and which strategy to choose depends on the context of a given project. For small-scale systems where the database schema is very stable, custom written JDBC code may be appropriate. It is lightweight in that it does not require any

third party infrastructure and does not involve the purchase of extra tools, since the JDBC APIs are part of the standard JDK.

Where database schemas are subject to change, and/or a system requires more complex mappings than simple types, some kind of mapping tool will pay off in terms of development time, robustness and maintainability. In particular, systems that extract the mapping into metadata that is manipulated using graphical tools are much easier to maintain and modify than hand written JDBC code.

EJBs are highly scaleable components that provide for standardised distribution and infrastructure, including transactional and security support. For large-scale enterprise systems they are frequently the best solution, being designed for component architectures where developers, assemblers and deployers may be working separately and where scalability is essential.

JDO has yet to prove itself in the marketplace, but offers a standard API for persistence that may span multiple data sources. Where it is necessary to integrate non RDBMS systems in to a persistence architecture, JDO may well offer an ideal way of abstracting out the client API from the underlying systems.

Combining strategies:

Although we have discussed the various persistence options as separate approaches, there are many ways in which they can be used together. For example, the Bi Modal Data Access pattern [iPlanet 2002] suggests that we can combine lightweight JDBC code (for example in JSP tag libraries) for read operations, with EJBs for transactional write and update operations. Tools such as O/R mapping tools or JDO implementations can be integrated with application servers, so that TopLink for example can be used as a persistence mechanism within an EJB container. JDO systems can similarly run in managed mode in a server and integrate with the server's infrastructure services. Figure 3 summarises the various options that might be considered for a hybrid persistence architecture.

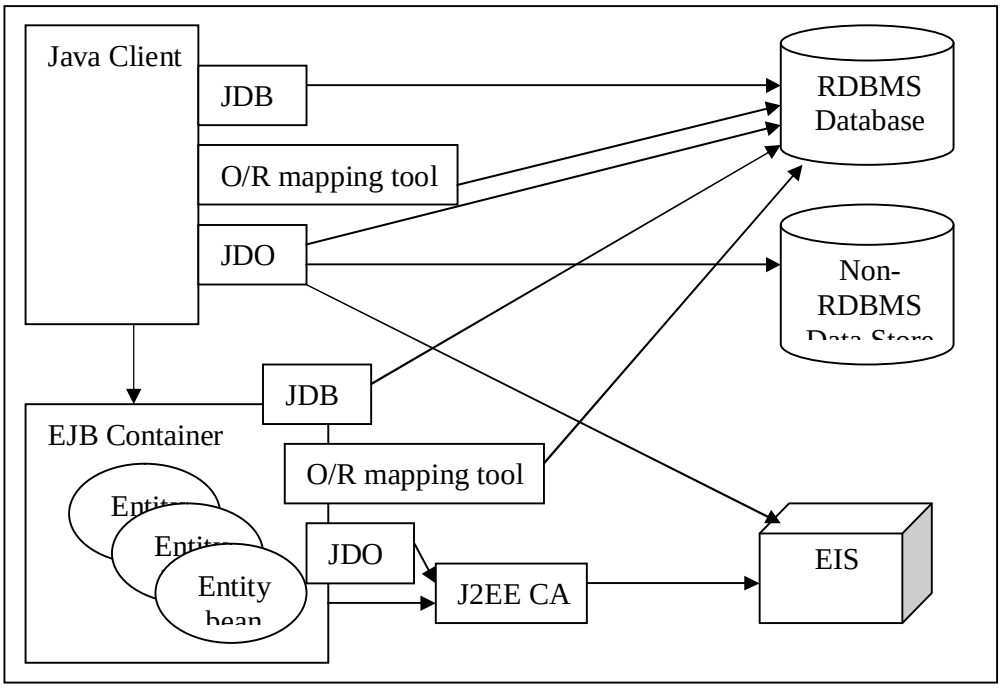

Figure 3: Options for persistence mechanisms in enterprise architectures

Deciding which combination of technologies will be right for a particular project will depend on issues such as cost, complexity of the required mappings, tools available for a particular platform, integration requirements and potential vendor lock-in. It is clear that none of these choices is simple, but it is also clear that the Java developer will have an increasingly powerful set of persistence tools available as the EJB and JDO specifications develop and mapping tool vendors increase their level of integration with other enterprise systems.

References

Ambler, S. 2000. *Mapping Objects To Relational Databases,* http://www.AmbySoft.com/mappingObjects.pdf.

Barry, D. and Stanienda, T. 1998. *Solving the Java Object Storage Problem*, IEEE Computer, November 1998 (Vol. 31, No. 11).

Brown, K. and Whitenack, B. 1996. *Crossing Chasms: A Pattern Language for Object-RDBMS Integration* in Vlissides, J., Coplien, J. and Kerth, N. (Eds.): Pattern Languages of Program Design 2, Addison-Wesley.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

iPlanet. 2002. *J2EE BluePrints*. http://www.iplanet.ne.jp/developers/ias-samples/jps1.1.1/docs/patterns/BimodalDataAccess.html.

Keller, W. 1997. *Mapping Objects To Tables - A Pattern Language,* Proceedings of EuroPLoP 1997. http://www.riehle.org/europlop-1997/.

Keller, W. and Coldeway, J. 1997. *Relational Database Access Layers – A Pattern Language* in Martin, R., Riehle, D. and Buschmann, F, (eds.) Pattern Languages of Program Design 3. Addison-Wesley.

Kruszelnicki**,** J. 2002. *Persist data with Java Data Object, Parts 1 and 2*, JavaWorld, March and April 2002.

Marinescu, F. 2002. *EJB Design Patterns – Advanced Patterns, Processes and Idioms*. Wiley.

Sun Microsystems. 2001. *Enterprise JavaBeans Specification version 2.0*, Sun Microsystems. http://www.java.sun.com/ejb.

WebGain. 2001. *TopLink 4.0 white paper*.