



Of Persistence and POJOs: Bridging the Object and Relational Worlds

by Rod Johnson and Jim Clark

The combination of Oracle TopLink and Spring's DAO layer offers a high-performance, productive approach to persisting Plain Old Java Objects.



Nearly every J2EE application needs to access one (or more) relational databases, so it's not an overstatement to say that one of the most important decisions you'll make when setting out to architect a J2EE application is how the application will access persistent data: Your persistence strategy not only can determine an application's performance, but can have a huge influence on the effort required to develop and maintain the application—and unless you make the right design decisions up front, it may be hard to revisit this part of the design after the application is finished.

In this installment of "Mastering J2EE Application Development," we'll look at two different (but complementary) frameworks that can work together or independently to help you overcome the complexity of accessing persistent data in J2EE applications, specifically:

- Oracle TopLink, a powerful object-relational *persistence framework* that provides object-oriented applications with a highly flexible, productive mechanism to access relational data
- The Spring Framework, a leading open source J2EE *application framework*, published under the Apache Software License. Spring provides services for all architectural tiers, but is particularly strong with respect to persistence by providing a consistent approach to data access. As you'll see, Spring's persistence services layer encompasses an implementation of the Data Access Objects (DAO) J2EE design pattern.

Using TopLink and Spring's DAO layer together can offer a high-performance, productive approach to persisting Plain Old Java Objects (POJOs)—that is, normal Java objects that are neither JavaBeans, nor Entity Beans, nor Session Beans—to relational databases.

This approach is an alternative to the traditional J2EE object persistence solution—EJB 2.x entity beans—which has proven disappointing in practice for many reasons, including a cumbersome programming model, poor developer productivity, and poor testability. Furthermore, EJB's query language (EJBQL) in its unadorned form—without proprietary extensions—is limited. Most important, however, entity beans place too many constraints on OO design by precluding the use of real OO inheritance, so they are not suitable for persisting sophisticated domain models. (POJOs, on the other hand, are especially useful for creating an object model of a specific problem domain.)

The approach to persistence presented in this article lets you persist POJOs without constraints on OO design, an approach which is, in our view, a better strategic choice than EJBs (see the "[On the Horizon: What about EJB 3.0?](#)" sidebar).

Although we focus in this article to some extent on TopLink and Spring, the technologies that they generally represent—O-R mapping, persistence layers implemented as DAO abstraction, separation of configuration via Inversion of Control (IoC)—offer lessons in architecture for J2EE developers. Such technologies as these are capable of persisting genuinely object-oriented domain models that capture important business concepts. They are often said to deliver *transparent persistence*—the ability for an object-relational mapping product to directly manipulate (using an object programming language) data stored in a relational database.

We'll highlight some of these capabilities and offer general guidelines, best practices, and basic strategies for J2EE persistence that you can use, regardless of any specific tool or framework.

Let's start with a quick overview of some of the challenges of developing applications that straddle the Java and relational database worlds.

The Need for Automating Data Access from Java

The JDBC API defines a standard way for Java code to talk to relational databases, but it's too low level for application developers to use productively. JDBC coding can be made more productive and less error-prone by using an abstraction layer over JDBC (such as Spring's JDBC packages or a good in-house JDBC framework, for example).

However, the developer—you—must still do a lot of low-level grunt work. For example, getting data into and out of the database requires setting values in JDBC PreparedStatements and extracting values from JDBC ResultSets.

Abstraction tools (such as iBATIS SQL Maps) that work at a slightly higher level than JDBC frameworks can automate much of the JDBC-to-SQL mapping. However, developers are still responsible for writing SQL, detecting when persistent objects are "dirty" (out of synch with the database), and writing the extensive Java code to store objects.

Sometimes working at such a low level is appropriate. Some applications don't lend themselves to greater automation than provided by such approaches. But most applications do. What's needed is the level of automation provided by sophisticated Object-Relational Mapping (ORM) tools.

Overview of O-R Mapping

You've probably heard of Object-Relational Mapping (ORM). You may already have used it; if not, you should certainly consider it for use in your Java or J2EE application.

ORM bridges the so-called Object-Relational impedance mismatch: the gulf between the object oriented model of a well-designed Java application (based on modeling real-world things and concepts) and the relational model in a database schema (based on mathematical approach to storing data). This gulf is surprisingly wide. A naive approach—simply mapping each class to a table—falls short in many respects. For example:

- Relationships between Java objects map to joins in the database, and can become expensive to traverse unless the mapping is optimized.
- Objects often participate in inheritance hierarchies, a concept that is foreign to the relational model.
- The standard relational model does not provide the extensible typing of a language such as Java.

"When setting out to architect an application, one of the key decisions is how it will access persistent data."

- If queries are written wholly in SQL, their relationship to Java objects may not be clear.

ORM is performed by a persistence framework, which knows how to query the database to retrieve Java objects, and how to persist those objects back to their representation in the database's tables and columns. Mappings are defined in metadata, typically XML files. Some popular tools provide graphical tools to make it easier to author mappings. (TopLink led the way here, with its excellent Mapping Workbench. See Figure 1. Also, see "[About Toplink](#)" sidebar for more information.)

With ORM, automation extends not only to mapping objects to database rows and columns, but also to detection of dirty objects and writing data back to the database with the least number of SQL updates—requirements that are appropriate for the majority of applications and which can be met more easily by using frameworks.

Key Benefits of ORM

ORM has a long history, and is not specific to Java. However, it has become particularly popular among Java developers. ORM has many purposes that can altogether add up to a huge productivity gain. Specifically:

- ORM eliminates the requirement to write SQL to load and persist object state. Although you still must write queries, a good ORM tool should make this much easier. And you will be free of the boilerplate coding associated with JDBC.
- ORM enables you to create an appropriate domain model without too many tradeoffs associated with the relational model. You can think mainly in terms of objects, rather than rows and columns.
- ORM can perform automatic change detection, eliminating an error-prone task from the development life-cycle.
- ORM can improve portability between databases by lessening dependence on database-specific SQL, which can be abstracted behind the ORM tool.

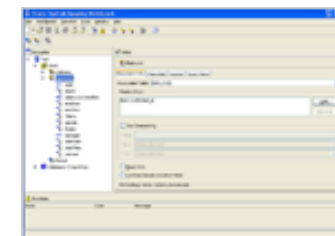


Figure 1: TopLink Mapping Workbench

(See end of article for larger image.)

The effect of ORM on application performance will depend on the scenario. For set-based update operations, ORM performance can be slower than straight JDBC, but in many cases, ORM can improve performance compared to hand-coded JDBC code, by enabling fine tuning of strategies such as lazy- or eager-loading of relationships that would be difficult to code by hand.

Sometimes, as it does with TopLink, ORM goes hand-in-hand with sophisticated caching mechanisms that can improve performance. The mileage you'll get out of caching will depend upon the nature of your application.

When and How to Choose ORM

"It is not unusual to see a saving of 30-50% in the amount of Java code that needs to be written through adopting an ORM solution instead of JDBC."

ORM has its skeptics. Especially in the .NET space, ORM is regarded with suspicion, and recently this skepticism has also been expressed loudly in the Java world.

Rather than theorize, let's draw on our own practical experience, across numerous J2EE projects. Used appropriately, ORM can slash development effort, and improve maintainability. It is not unusual to see a saving of 30-50% in the amount of Java code that needs to be written by adopting an ORM solution, rather than JDBC. This is a big win, because all the effort saved can go into implementing business features. It is simply irresponsible to ignore the potential benefit available through effective use of ORM in the right applications, in the right environment.

Negative experiences with ORM tools often result from trying to use ORM when it isn't appropriate. ORM is likely to produce good results as long as you don't force a ORM when it doesn't fit. Here are some basic guidelines:

Understand your target database. Don't think that using ORM lets you ignore SQL and your database's locking model. O-R mapping is a tool to make doing what you want to do easier—it doesn't free you from the need to understand what it is you want to do. (Not appreciating the nuances of the database and SQL is probably the main cause of disappointing results with ORM.)

Don't be afraid to use SQL if necessary. It works well in many scenarios. A good ORM product such as TopLink will let you issue SQL queries. But sometimes you may need to perform set-based updates or invoke stored procedures.

Do your research before committing to an O-R mapping product. Not all ORM products are created equal. Be sure to evaluate two or three alternative products by developing a proof of concept that reflects your needs. This exercise will help to ensure that your use of ORM is appropriate on performance grounds. As with enterprise development in general, it is vital to mitigate any performance risks early in the project lifecycle. Also, your ORM tool's mapping function shouldn't come with a lot of overhead.

Know when to use ORM. ORM works particularly well for OLTP applications that update entities individually and perform set-based operations infrequently, such as applications that update customer records and their associated orders, individually.

And, when not to use it. ORM isn't always a good fit. It's not appropriate for:

- Applications that perform frequent bulk updates to numerous records.
- OLAP applications, because databases typically provide valuable native programming mechanisms. (In general, J2EE may not be a good fit for OLAP.)
- Database or operational environments replete with hand-coded SQL and stored procedure calls as the sole mechanism for retrieving and updating data. In such cases, a JDBC-based approach may be the best choice; iBATIS SQL Maps also shines in such scenarios. (Note however, that a good ORM product, such as TopLink, can work with most existing schemas and with stored procedures, so you might still consider ORM in such scenarios.)
- Applications that are better served by straight SQL-based approaches, such as applications that rely heavily on business logic already coded in the database, or enforced by integrity constraints, and that give users a "window on data," enabling users to edit data. In applications such as these, ORM has less to offer because *objects*, in general, have less to offer—beyond an illusion of object-orientation, little may be gained by modeling database tables as domain objects.

There are several good ORM tools available today, including both commercial (TopLink, for example, the oldest Java ORM product and several JDO implementations) and open source (such as the Hibernate framework, or JDO implementations). Simply choose one of them—there's no need to roll your own. (In the past it was common to develop in-house ORM frameworks, but these days, it doesn't make sense: it's an expensive undertaking whose results are never as good as generic commercial or open source solutions.) Let's take a closer look at using TopLink for a minute as an example of using an ORM framework, and the productivity benefits that accrue.

An Example of Working with an ORM Tool

The TopLink Mapping Workbench lets you define mappings, which are persisted in XML files or Java classes. (Mapping files are human readable and can be edited.) The Mapping Workbench reads database metadata to provide assistance in identifying database column names to map, handling foreign key relationships, helping to eliminate mapping errors.

There are two major approaches to querying in ORM products—string-based query languages (such as Hibernate HQL or SQL itself), and API-based expression languages, such as TopLink Expressions language, a powerful expression language for querying. Although expression languages may (superficially) seem more verbose, they can better guarantee correct semantics.

Here's an example of a TopLink Expression-based query in Java:

```
ExpressionBuilder builder = new ExpressionBuilder();

Expression exp = builder.get("address").get("country").equal("USA");
exp = exp.and(builder.get("address").get("pCode").equal("27615"));
exp = exp.or(builder.get("address").get("country").equal(""));

session.readAllObjects(Employee.class, exp);
```

As shown in Figure 2 and Figure 3, the TopLink Mapping Workbench also lets you build queries using a graphical approach.

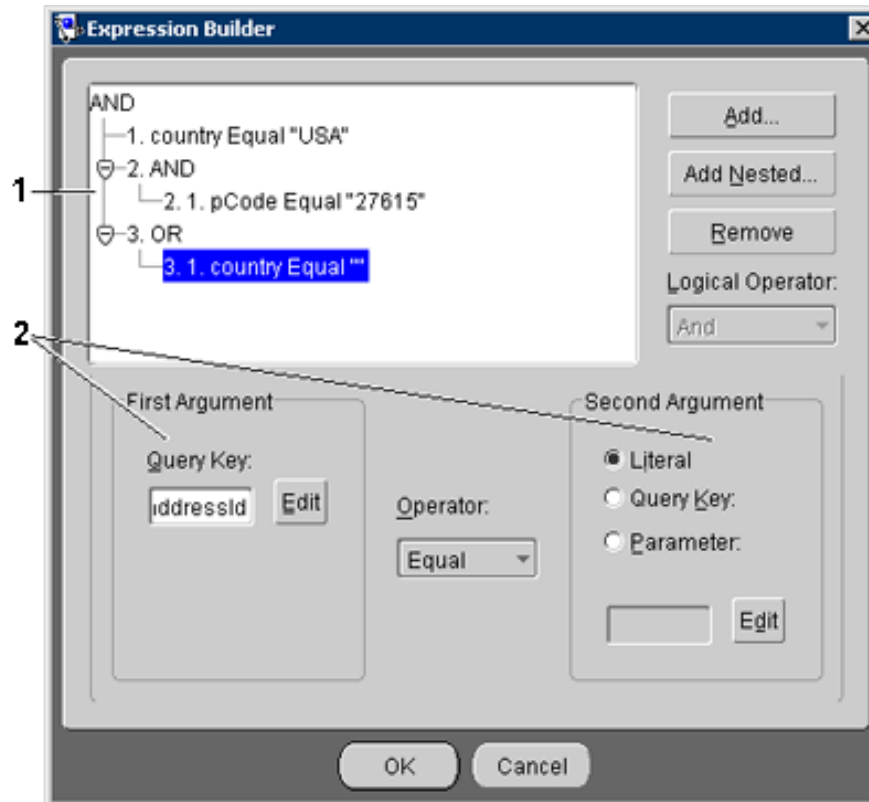


Figure 2: Oracle TopLink Expression Builder

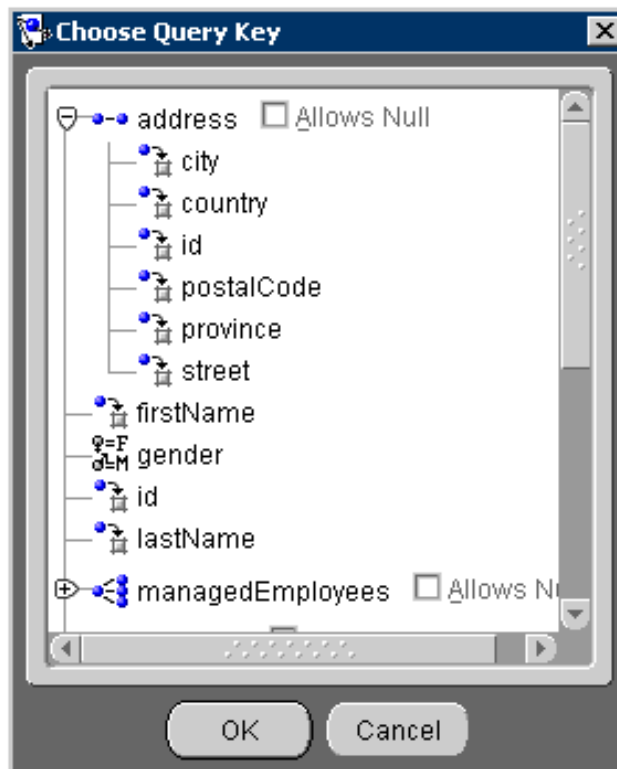


Figure 3: Choosing a query key using the TopLink Expression Builder

TopLink Expressions are very capable, and support advanced relational concepts such as GROUP BY. However, you can also use SQL within TopLink if you need to 'drop down to the metal.'

As you can see from these screenshots, TopLink Mapping Workbench lets you define Java-based queries while looking directly at your data model, without writing Java code. The queries are saved in metadata.

Regardless of the specific ORM tool you use, you must master the product, and your database.

The Spring Framework, ORM, and Persistence

The Spring Framework is a leading open source J2EE application framework that provides services for all architectural tiers (see "[About the Spring Framework](#)" sidebar for more information).

Unlike a single-tier framework (such as Struts), Spring is an *application framework* designed to facilitate a clean, layered approach to application architecture. Spring does not impose this architecture, but it makes it easier to follow good practice. The typical Spring architected application consists of the presentation layer, services layer, DAO interface and implementation layer, and a persistent domain objects layer.

In terms of object persistence, it's the DAO interfaces layer that drives data access, implementing queries and other persistence operations. Let's take a closer look at that layer, first in generic terms, and then in terms of how Spring's DAO layer can be used in conjunction with an ORM tool (such as TopLink).

Benefits of DAO Abstraction Layer

A well-designed DAO interface can wholly conceal persistence technology specifics from service layer objects. For example, you can implement the same DAO interface with TopLink or Hibernate, without impacting calling code.

This is good, because it lessens lock-in to a particular ORM tool, and provides a degree of future proofing. Data access concepts won't change in the next few years, but the technologies that implement DAO will. Even more importantly, it means that you can easily test service layer objects without access to a database, by stubbing or mocking DAOs.

As a general mechanism (along the lines of Eric Evans' notion of "Repositories" (see [Domain-Driven Design](#) there is nothing Spring-specific here), DAO interfaces usually offer these types of methods:

- *Finders* for retrieving persistent object instances. Finders are implemented using the O-R mapping tool's query interface.
- *Save methods*, for persisting new objects
- *Delete methods*, for deleting persistent objects from the database.
- *Aggregate function* methods for counts or other aggregate functions. Querying the database for aggregation is usually much faster than iterating over large numbers of persistent entities.

A Closer Look at Spring's DAO Layer

In a typical Spring architecture, DAOs are effectively Strategy Pattern objects, isolating service layer objects from the technology specifics of obtaining and saving persistent objects.

Spring supports a range of ORM tools and persistence frameworks, including (but not limited to) TopLink, Hibernate, JDO (Java Data Objects), iBATIS SQL Maps, and Apache OJB (ObjectRelational Bridge).

Spring provides a consistent approach to working with all these tools, but doesn't attempt to completely abstract the ORM tool, because doing so would be counterproductive. Thus, for example, implementations of DAO interfaces will use the native query API of the ORM tool: Spring does not attempt to create a query abstraction, which would be less powerful than that of leading ORM products.

How Spring Can Leverage any ORM

For DAO interfaces to be independent of the underlying persistence technology, we need to solve some crucial issues. Spring provides important help here, especially in the areas of exception handling; resource acquisition and release; and transaction management.

Spring exception handling: Two key features of Spring's exception handling enable it to remain independent of the underlying persistence technology. First, it uses a class-based hierarchy of generic data access exceptions (Figure 4 shows the most important classes in the hierarchy) that can leverage any underlying ORM's exception mechanisms. The exception hierarchy can easily be extended. Thus, for example, you can write service layer code to handle `DataIntegrityViolationException` (violation of a constraint such as a primary key constraint), without concern as to the data access tool in use, which may be JDBC, TopLink, JDO, Hibernate or another supported framework. This hierarchy is even extensible without modifying Spring code, if you wish to add mappings to error conditions unique to your database or persistence tool.

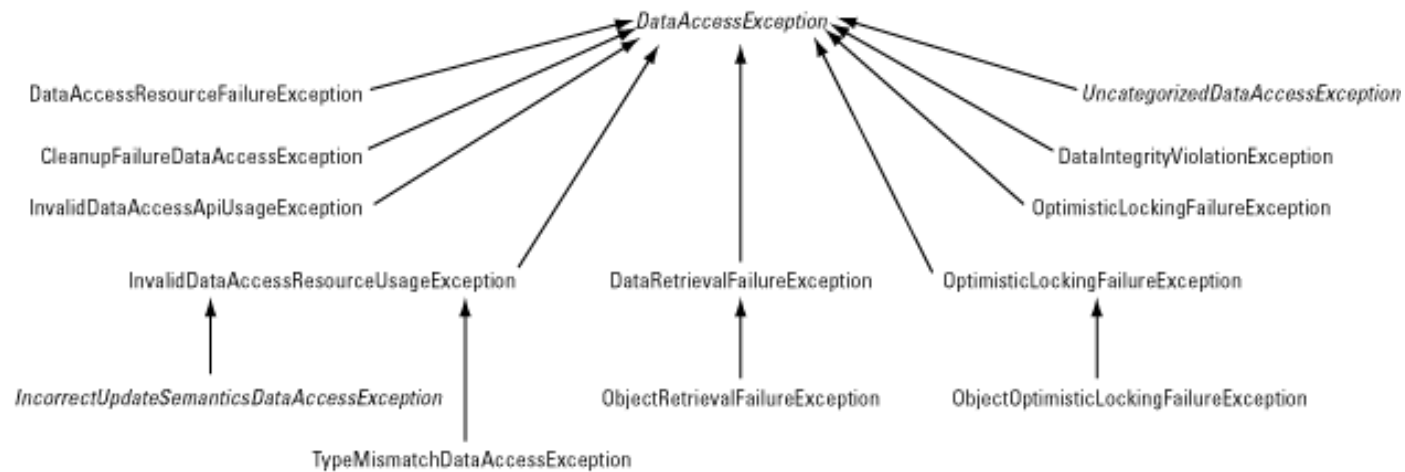


Figure 4. UML diagram of Spring's exception class hierarchy

Secondly, Spring's exception handling approach uses unchecked exceptions. Using unchecked exceptions works particularly well in conjunction with an exception hierarchy, because it is only necessary to catch a particular subclass that may be recoverable. Catching subclasses of checked exceptions is less useful, since the base class still needs to be caught. (Note that JDBC was always something of an odd API out, in terms of using checked exceptions: TopLink and JDO, for example, both use unchecked exceptions. In version 3, Hibernate will switch from checked to unchecked exceptions.)

Resource acquisition and release: One of the major challenges of working with any data access technology is that resources such as JDBC connections or ORM sessions must be acquired and released correctly even in the event of errors.

Spring solves the problem of resource acquisition and release by using a callback approach. The framework is responsible for acquiring and releasing resources, while the application developer implements code in a callback method to perform the necessary persistence operation with the desired resource. Thus application developers focus on the unique needs of their application, rather than writing plumbing code.

Helper classes that provide this functionality are termed templates, and Spring provides a consistent approach across many APIs where such resource acquisition and release is a problem, such as JDBC, JNDI and JMS.

(An ORM *session* is a transactional unit-of-work managed by the ORM tool. A session contains a set of objects associated with the transaction, some of which may have been modified during the course of the transaction, and for which the ORM tool generates SQL to update the database. Thus, session management is central to data integrity.)

Transaction management and thread binding: Spring's resource acquisition and release strategy is integrated with its transaction management support. It's important to get the same resource for every operation in the same transaction. While JTA and JCA may provide support for this in a full-blown J2EE environment (depending on the persistence tool), Spring provides support for this in any environment, including J2SE environments. The necessary Spring configuration is also simpler than JCA configuration.

Spring does this by binding resources such as JDBC connections or TopLink or Hibernate Sessions to the current thread. Many application developers have implemented such support in their own applications. However, the Spring solution not only has the merit of being far more sophisticated than typical in-house solutions (with support for transaction suspension and many other complex-to-implement value adds), but eliminates the need to write and maintain such custom code. Again, generic concerns are best given a generic solution.

Convenience Classes and Methods

Spring provides convenience classes to simplify working with each supported persistence tool.

For example, the JDBC, TopLink, and Hibernate support classes provide query and save methods that reduce such operations to a single line of code, as in the following:

Hibernate query using the Spring HibernateTemplate convenience class:

```
List customers = hibernateTemplate.find("from Customer c where c.age > ?", age);
```

TopLink query using the Spring TopLinkTemplate convenience class:

```
List customers = toplinkTemplate.findByNameQuery(Customer.class, "findAllCustomersOlderThan", args);
```

Such convenience methods can significantly reduce the amount of data access code required in an application using the native API of the persistence tool and dealing with resource acquisition and release.

Spring and TopLink are a particularly good match. TopLink is a complex product, so the consistent architectural approach and "template" recipe brought by Spring can provide users with useful guidance.

An Example of Persistence in Action: Spring and TopLink Integration Project

Now available on OTN, the Spring/TopLink integration builds on the consistent architectural pattern established by Spring's integration with other ORM frameworks. The Spring-TopLink integration code was written by the Oracle TopLink team, but is likely to be donated by Oracle to the Spring project. Both Oracle and the Spring community plan to support users of the integration.

Spring-TopLink integration enables you to enjoy a consistent programming model whether you use TopLink inside or outside an application server. The Spring TopLink integration includes:

- Convenience classes, such as TopLink template, that make many TopLink operations very easy and provide simplified model for implementing more complex operations by eliminating resource lookup.
- Exception translation utilities, that can translate TopLink exceptions into Spring's DataAccessException hierarchy.
- Integration with Spring transaction management A proof-of-concept application, PetClinic, demonstrates the integration and will help get you started with Spring and TopLink. Although PetClinic is a simple application, it illustrates some important best practices, and can serve as an architectural template for much sophisticated use of Spring and TopLink.

On the Horizon: What about EJB 3.0?

The Enterprise JavaBeans (EJB) 3.0 specification, currently being developed through the JCP as JSR-220, will include a POJO persistence specification that will effectively render the existing entity bean model obsolete. This initiative has also led to some uncertainty in the O-R mapping space.

Nonetheless, if your application can benefit from O-R mapping, you should adopt an ORM tool now, and enjoy the resulting productivity gains. If you choose a leading product, such as Oracle TopLink, Hibernate, or Kodo JDO, you can be sure that the vendor will provide a migration strategy to JSR-220 POJO persistence—you may well be using the same tool now in two years time, but simply through a different API. (Migration from EJB 2.x entity model to the JSR-220 POJO persistence will be more difficult.)

Furthermore, the EJB 3.0 specification is not due for release until early 2006—in the meantime, it's vital to have a viable, productive O-R mapping solution.

By default, the PetClinic runs against HSQL (HypersonicSQL, a simple, pure Java database that is useful for testing or for applications with very simple data access requirements). But with straightforward reconfiguration in the TopLink layer, you can use Oracle or another full-featured database (instructions are available in the download for reconfiguring PetClinic to use Oracle.)

When you use Spring with TopLink, be sure to:

- Use named queries. This has long been recommended as a best practice from TopLink consulting.
- Use the TopLinkTemplate class to obtain and use TopLink sessions, rather than write your own code to manage TopLink sessions. This will save you time and effort and will ensure consistent error handling in your application.
- Follow the normal Spring architectural practice, and hide your use of TopLink behind the DAO interfaces.

Next Steps

- **Download the Spring/TopLink Integration project**
- **Download Oracle TopLink 10g (10.1.3)**
- **Read J2EE without EJB, Johnson and Hoeller**, Wrox, 2004. This discusses lightweight J2EE architecture, emphasizing Spring and ORM.
- **Read the classic text on object-orientation, "Design Patterns,"** by Gamma, Helm, Johnson and Vlissides, (Addison-Wesley, 1995), for more information about the Strategy design pattern.

Conclusion

Persistence is vital to success in J2EE projects. Hopefully this article has convinced you of the value proposition of ORM. If your project can benefit from ORM, then your productivity will likely get a boost.

The Spring Framework is an application framework that provides services for all tiers of Java/J2EE applications. In this article we have focused on Spring's approach to persistence, but this is in fact just part of what Spring provides. Spring provides a consistent architectural style for J2EE applications accessing persistent data. Spring integrates with all leading O-R mapping technologies, including Oracle TopLink. Oracle's commitment to Spring-TopLink integration is good news for both the Spring and TopLink communities.

If you're a TopLink user, you will appreciate the simplification and consistency that this can bring to your coding. Yet, you won't need to sacrifice any of the power of TopLink. If you've a Spring user, you have gained the choice of TopLink for your ORM requirements.

About TopLink

TopLink is probably the most mature ORM framework. It was first developed in Smalltalk in 1994. Since 1997, its main focus has been Java. Today TopLink can be used inside or outside a J2EE application server. While it is now owned and supported by Oracle, TopLink works with any major database, not just Oracle. It works in all J2EE application servers—or, indeed, outside an application server altogether.

TopLink provides a wide range of ORM functionality. Like all good ORM tools, TopLink enables you to persist POJOs (Plain Java Objects), which have minimal dependence on TopLink APIs. TopLink provides a wide range of advanced mappings, including support for inheritance and BLOB types (with streaming). It also provides sophisticated support for optimistic locking, which can produce throughput gains if used appropriately.

It is important to note that TopLink is not purely an ORM tool but also provides an integrated caching solution (including cache coordination across clusters), rich query support, O-X capabilities, several performance features (such as "indirection") that optimize interactions with the database, and transaction management.

With Oracle JDeveloper 10g (10.1.3) developer preview you get a single IDE that lets you define your toplink mapping, and visually build your JSF and JSP pages, in addition to writing your Java code. Oracle JDeveloper integrates the TopLink Mapper to help developers visually map Java objects to database tables using the powerful TopLink persistence layer. Developers using TopLink as the persistence architecture for their custom Java classes can leverage the common data-binding features in the ADF Model layer to simplify building JSP, uiXML, and Swing application user interfaces.

Over the years TopLink has been recognized as a leader in providing persistence services for Java systems. Here are some additional Oracle TopLink resources:

- [A Brief History of TopLink](#)
- [Overview of TopLink Caching and Locking](#)
- [Introduction to TopLink Object-XML Mapping](#)
- [Preparing for EJB 3.0](#)
- [Upcoming TopLink Online Seminars](#)
- [More TopLink Resources](#)
- [Oracle Application Server 10g TopLink Documentation](#)
- [Building JSF/TopLink applications with Oracle JDeveloper](#)

About Spring

Spring is a multi-layered application framework—something of a "framework of loosely-coupled sub-frameworks." Among its many features, some of Spring's key services include:

- An Inversion of Control (IoC) container (Spring Core package) that supports Dependency Injection, a powerful technique pioneered by Spring for configuring POJO-based applications. Spring is the most popular and powerful of a number of frameworks in the IoC space.
- A pure Java AOP (Aspect-Oriented Programming) framework (the Spring AOP package) that enables crosscutting functionality to be applied to POJOs, which enables sophisticated support for declarative transaction management in a range of environments.
- A flexible web MVC framework. Spring also integrates out-of-the-box with popular web tier solutions such as Struts, WebWork, Tapestry, or JSF.
- A common approach to persistence (the Spring DAO package).
- Support for remoting, working with EJBs, JMS, and other important enterprise functionality (Spring Context package).
- Integration with numerous third-party products, including the Quartz scheduler and Velocity template engine.

Spring's scope is much broader than just this short list, however. Here are some resources for additional information:

- [Spring home page](#).
- "[Introducing the Spring Framework](#)," by Rod Johnson, provides a complete overview of the core concepts.
- "[Thread-safe Webapps using Spring](#)" by Steven Devijver. Discusses how Spring manages transactional resources during data access operations.


Rod Johnson is CEO of Interface21, a consultancy specializing in expert J2EE and Spring Framework services. He is an enterprise Java architect with extensive experience in the insurance, dot-com, and financial industries. He was the J2EE architect of one of Europe's largest web portals, and he has worked as a consultant on a wide range of projects. He is actively involved in the Java Community Process as a member of the JSR-154 (Servlet 2.4) and JDO 2.0 Expert Groups. He is the author of the best-selling "Expert One-on-One J2EE Design and Development" (Wrox, 2002), and "J2EE without EJB" (Wrox, 2004) and has contributed to several other books on J2EE since 2000. Rod is prominent in the open source community as co-founder of the [Spring Framework](#) open source project, which grew out of code published in "Expert One-on-One J2EE Design and Development."

Jim Clark (james.x.clark@oracle.com) is a member of the OracleAS Solution Architects Team. He specializes in J2EE/Toplink development and is currently driving Oracle's effort to integrate TopLink and Spring. Other current projects include using AOP techniques to enhance Object persistence.

Copyright © 2005, Oracle. All Rights Reserved.

The screenshot displays the Oracle TopLink Mapping Workbench interface. The title bar reads "Oracle TopLink Mapping Workbench". The menu bar includes "File", "Workbench", "Selected", "Tools", "Window", and "Help". The toolbar contains various icons for file operations and help. The left pane, labeled "Navigator", shows a tree structure with "Test" expanded to "demo", which contains "Address" and "Employee". The "Employee" class is selected, showing its attributes: "addr", "empld", "employeeCollection", "endDate", "endTime", "fName", "gender", "lName", "manager", "startDate", "startTime", and "version". The right pane, labeled "Editor", shows the "Employee" class configuration. It has tabs for "Descriptor Info", "Class Info", "Queries", and "Query Keys". The "Descriptor Info" tab is active, showing "Associated Table: EMPLOYEE". Below this, the "Primary Keys" section contains a list with "EMPLOYEE.EMP_ID" and buttons for "Add..." and "Remove". At the bottom, there are checkboxes for "Use Sequencing:" (unchecked), "Read-Only" (unchecked), and "Conform Results in Unit of Work" (unchecked). The "Use Sequencing:" section includes fields for "Name:", "Table: <none selected>", and "Field: <none selected>".

Confirm Results in Unit of Work
Refreshing Cache Options (Advanced)

 Problems

Node	Code	Message

Figure 1.