Object/Relational Mapping 2008: Hibernate and the Entity Data Model (EDM)

Elizabeth (Betty) O'Neil
Dept. of Computer Science
University of Massachusetts Boston
Boston, MA 02125
eoneil@cs.umb.edu

ABSTRACT

Object/Relational Mapping (ORM) provides a methodology and mechanism for object-oriented systems to hold their long-term data safely in a database, with transactional control over it, yet have it expressed when needed in program objects. Instead of bundles of special code for this, ORM encourages models and use of constraints for the application, which then runs in a context set up by the ORM. Today's web applications are particularly well-suited to this approach, as they are necessarily multithreaded and thus are prone to race conditions unless the interaction with the database is very carefully implemented. The ORM approach was first realized in Hibernate, an open source project for Java systems started in 2002, and this year is joined by Microsoft's Entity Data Model for .NET systems. Both are described here.

Categories and Subject Descriptors

H.2 [**Database Management**]: H.2.1 Logical Design *Data models, Normal forms, Schema and subschema* H.2.3 Languages *Database (persistent) programming languages* H.2.8 Database Applications, D.2[**Software Engineering**] D.2.2 Design Tools and Techniques *Object-Oriented Design methods* D.2.11 Software Architectures *Data abstraction*

General Terms

Design, Languages, Algorithms

Keywords

Object-relational mapping, impedance mismatch, data model, persistence, schema mapping, web application, Hibernate, Entity Data Model

1. INTRODUCTION

Programmers strongly prefer to work with persistent data held (for the moment, anyway) in program objects, rather than use SQL directly for data access, even though this means working around the famous "impedance mismatch" between tabular data and object state. Object/Relational mapping systems bridge this mismatch, by whisking data to and from a relational database to appropriate objects, based on O/R mappings. O/R mappings map object schemas (class diagrams, etc.) to database schemas, part of the technology of last year's SIGMOD keynote talk [5] by Phil Bernstein on Schema Mapping. As Phil mentioned, there is a recently developed schema language in Microsoft's ADO.NET called the Entity Data Model [10] (for release in 2008); this will be described more fully in the current talk. A comparable model and system was developed by the open-source Hibernate project, founded and led by Gavin King starting in 2002, after his frustration as a software developer with the "heavy-weight" Entity Java Beans (EJB) of the older Java Enterprise platform. King describes the Hibernate system in an excellent book on the subject [3]. The current EJB specification, EJB 3.0, uses the direct descendent of Hibernate 3.0, known as Java Persistence Architecture (JPA).

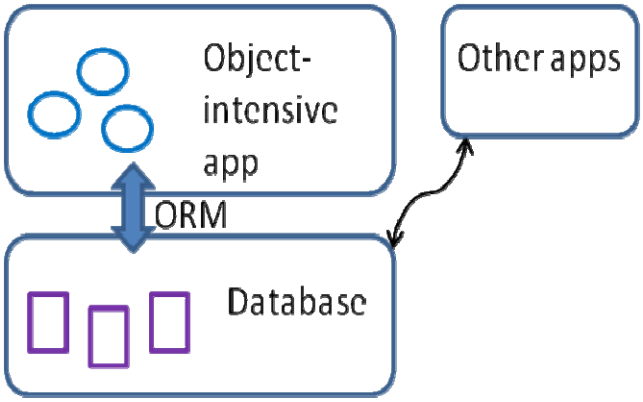

Figure 1. ORM in use in one of many apps using the database.

I will call the common ideas of Microsoft's Entity Data Model and the Hibernate model (and JPA) simply the "Entity Model", since the entities correspond to the objects, making them the centers of attention. The Entity Model is a refinement of the entity-relationship (E-R) model of Chen [6], and the extended E-R model (EER) that incorporates inheritance/generalization hierarchies. The Entity Model sits between the object world of applications and the underlying database(s) where all persistent data is stored, as shown in Figure 1. Typically, other applications access the database as well, as the database represents shared enterprise data.

Both Hibernate and EDM provide a full object/relational mapping system, and have GUI tools to help with development. For simplicity, I'll be covering Hibernate and EDM, and ignoring JPA, since it is so close to Hibernate and largely derives from it. As the standardized offering, JPA is advancing in adoption by all the major Java enterprise application server products. Hibernate and its JPA implementation is part of JBoss [7], the first

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.
ACM 978-1-60558-102-6/08/06.

application server with this capability. OpenJPA, originally from BEA's Kodo product, is now an open-source project at Apache, and is available for BEA's WebLogic Server [4] and IBM's WebSphere [8]. Oracle's TopLink 11g supports JPA [11].

With the Entity Model, the mapping from application objects to tables is made in two hops, with the Entity Model in the middle, apparently a more complex task than a single mapping. Why would this be a good idea? The answer is that this Entity Model is useful to the practitioners for designing and implementing real systems. It abstracts away some of the nitty-gritty details of database table setup and pastes over some of the deficiencies of relational tables, such as the difficulty of expressing a generalization/inheritance hierarchy. It provides guidance towards workable database schemas.

With these O/R mapping systems and their Entity Models, a programmer is encouraged to think in terms of entities and their relationships. The system takes over all the details of handling relationships at runtime, delivering object graphs for complex objects, for example, ready for programmatic use. The system automatically tracks updates made to the objects, and performs the necessary SQL insert, update, and delete statements at commit time. Thus the business logic programming can be done in the comfort of object-oriented languages, usually Java or C#, with transactions delineating the discrete actions of the application at the object level. The objects we are discussing here are a subset of all the objects in the application, the ones (temporarily) representing persistent data. They are often called persistent objects, but this name causes some confusion with object serialization (the Java/C# object persistence mechanism), so I will use another common name, domain object.

Of course there are differences between the Hibernate and EDM systems, but the main thrust of what I present is their common features, a convergence of technology relevant to the database community. In fact, the EDM system so far is available only in beta release (free), so it is still a work in progress and lacks some needed features.

For simplicity, a single database server is considered, although both systems can handle distributed transactions with the help of JTA/DTC. The Hibernate platform is in use for systems involving up to hundreds of entities and tens of thousands of users, that is, it is scalable up to the point that the database(s) involved is/are overloaded. This approach is relevant to most dynamic Internet sites, all but the very largest, plus most non-web database applications with significant user interaction.

2. THE ENTITY MODEL

An entity has attributes as in E-R, called properties in the Entity Model, and a unique identifier, often a "surrogate" key, meaning one whose value is not important to the application aside from its use as an identifier. For example, in this approach a SSN would not be used as an id but rather as a property. We will assume this in what follows, for simplicity. Natural keys can also be used. See Ambler [2] for discussion of key choices. The unique identifier is persistent, like its corresponding database primary key.

Relationships boil down to the binary N-1, 1-1 and N-N cases. A ternary relationship is not directly supported by the Entity Model, but can be expressed by promoting its links to entity instances.

As is well known, databases are bad at modeling inheritance hierarchies. There are several ways to do it, none completely satisfactory. The Entity Model abstracts the concept, and then provides alternative implementations, selected by configuration. Today, the programmer needs to craft specific directions in XML or provide code annotations to get a class inheritance hierarchy properly mapped to the chosen database solution. Both Hibernate and EDM can follow the most common alternatives, known as table-per-hierarchy (one table for all variants) and table-per-type (one table for base plus one table per subclass)

3. THE PIZZA SHOP EXAMPLE

This example is available at www.cs.umb.edu/~eoneil/orm. This website contains side-by-side implementations, using Hibernate and Microsoft EDM, of a simple system with four database entities related by an N-N relationship and an N-1 relationship (one entity has no relationship to the others and is ignored in the figures). See Figure 2. It is a system for ordering free pizza to be delivered to a specified room number in a dormitory. The "student" user has choices of pizza size and toppings, so each order has one size and a set of toppings. The "admin" can add and delete topping choices and sizes, mark pizza orders as ready, finish off a day and start another, etc. Although the system is simple, it is implemented with the layered architecture of larger applications, with a transactional service layer called by the presentation layer, which contains all the user interface code.

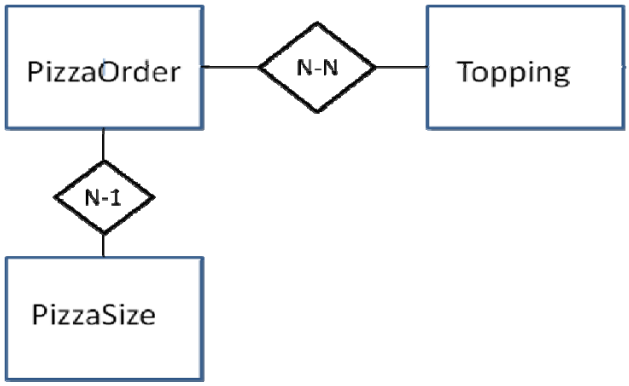

Figure 2. E-R Diagram for Pizza Shop Database

Figure 3 shows the database schema for the system. The N-1 relationship is simply represented by the sizeid foreign key in the pizza_order table. The N-N relationship needs a "link table" with foreign keys to the two related tables, pizza_order and topping.

The corresponding entity model is shown in Figure 4. The relationships are now reduced to annotated lines between the entities, as is also commonly done in UML [12] for object models. The link table is considered an implementation detail and suppressed from the top-level diagram. Its existence can be surmised from the cardinality markings "1..*" and "0..* having stars at both ends of the relationship line. These markings denote (1,N) and (0,N) multiplicity, respectively. PizzaSize has the default multiplicity of (1,1) in its relationship and thus has no marking on its end for the N-1 relationship.

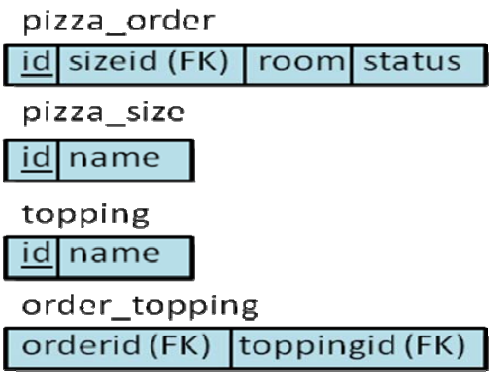

Figure 3. Database schema with link table order_topping

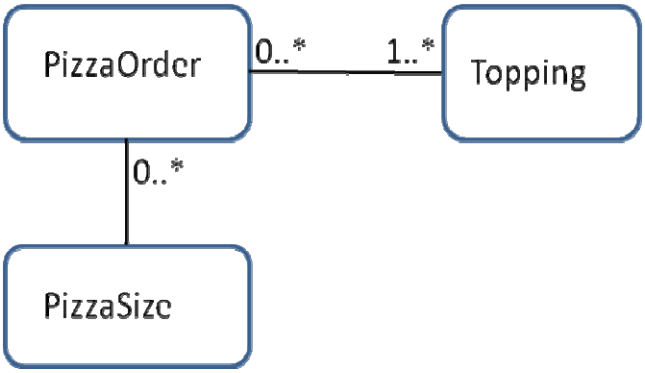

Figure 4. Entity Model

During execution, a pizza order is a small object graph of domain objects: the main PizzaOrder object referencing a PizzaSize object and a collection of Topping objects. Here is a quick example of application code working with the domain objects. To find the size name for a PizzaOrder order, just "dot" through the to-1 relationship:

```
order.getSize().getSizeName() // Hibernate/Java
order.Size.SizeName           // EDM/C#
```

We will also look at an example of iterating through the toppings. To get orders from the database, we use the Hibernate Session or EDM ObjectContext "context" object to create a query that can return PizzaOrder objects, as follows:

```
Query orders =                  //Hibernate
  context.createQuery("select o from
    PizzaOrder o " +
    "where o.roomNumber = " + roomNumber
      + "and o.day = " + day);

ObjectQuery<PizzaOrder> orders =   //EDM
  new ObjectQuery<PizzaOrder>(
    "select value o from
    PizzaEntities.PizzaOrder as o " +
    "where o.RoomNumber = " + roomNumber
       + "and o.Day = " + day, context);
```

We see that we are not really giving up SQL query power by using O/R Mapping. SQL itself is mapped into the object world, allowing joins (inner and outer), ordering, group by, polymorphism (across subclasses), and prepared statements. The query results can consist of domain objects or other program objects. If the object query language is still too restrictive, the underlying connection to the database can be used for direct SQL. Such direct SQL is commonly used for batch updates and database reloads.

4. THE PRESENTATION, SERVICE, AND DATA ACCESS OBJECT (DAO) LAYERS

In serious applications, the code is organized into three layers, the presentation, service, and DAO (data access object) layers. See Figure 5. The presentation layer code calls the service layer methods, and the service layer code calls the DAO layer methods. Domain objects are passed as arguments and return values of these calls; they are used in all the layers. Domain objects carry data around but don't have to be "dumb" data carriers, that is, they can have specialized methods to help with the needed work. Presentation-layer code implements the user interface (UI). It calls the service layer for all actions related to persistent data. The presentation layer for the pizza project is provided in two versions for each O/R mapping framework, a line-oriented UI and a web application with server-side scripting in JSP 2.0/ASP.NET.

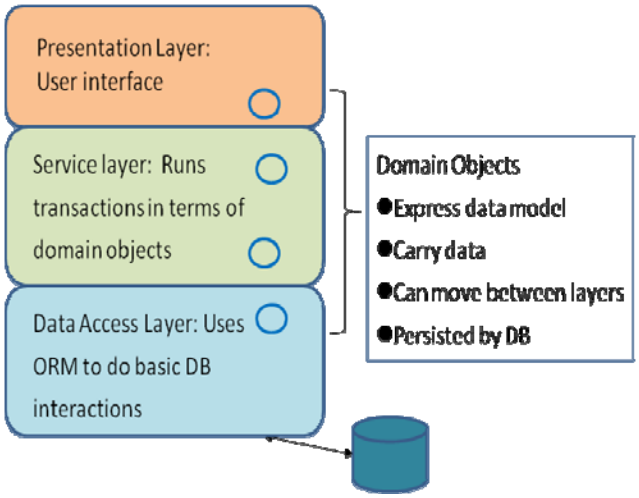

Figure 5. A Layered Database Application

Service-layer code implements the basic transactional actions by calling the DAO layer to get domain objects, working with them, and then possibly updating them with the help of the DAO. This layer is also called the business layer, because it implements the business model actions. Here we see methods makeOrder, getOrderStatus, allToppings, allPizzaSizes, addTopping, etc., together constituting the "service API" (applications programming interface) called by the presentation layer. These names are capitalized for C# (MakeOrder, etc.) to follow coding conventions. The service-layer code starts up a transaction around the needed actions and ultimately commits or aborts it.

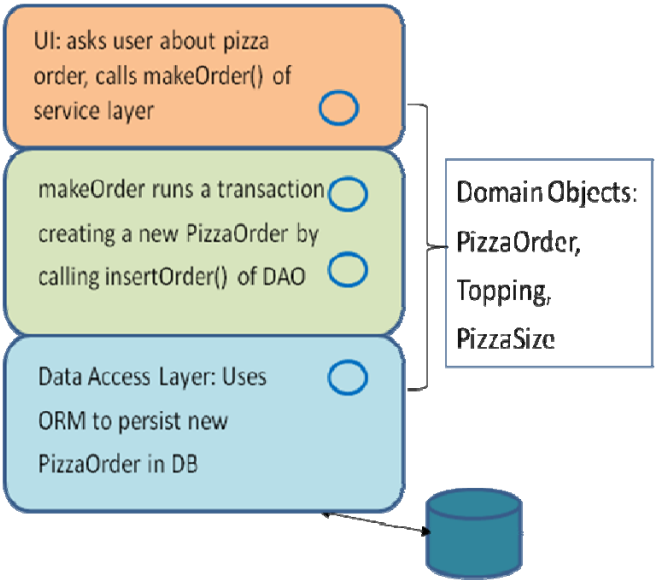

Figure 6. Calls down the layers in Pizza Shop

DAO code inherits an ongoing transaction from its service-layer caller, and works with the O/R framework to get and update domain objects. DAOs use object queries (as shown in the code above) or primary-key lookups to read data, or less commonly, direct SQL. Figure 6 indicates some of the calls involved in making a new pizza order. First, in a previous time (and transaction) to what is shown here, the presentation layer calls the service layer to get Topping and PizzaSize objects to show the user the possible pizzas to order. After the user decides, the presentation layer calls makeOrder(), and the service layer runs a transaction. During the transaction, the service layer calls the DAO layer, and the new pizza order object graph is persisted. In particular, a new row is added to pizza_order and one row for each topping is added to order_topping. If the topping has meanwhile disappeared, the DAO access fails and the service layer aborts the transaction.

5. THE ENTITY CONTEXT

The execution environment provided by the O/R Mapping platforms is delivered by the Hibernate Session and the EDM ObjectContext, seen as the "context" object earlier in DAO code. Let us call this common idea the entity context. The entity context provides a private cache of objects for the application execution in one unit of work (thread), with at most one object instance for each entity id. See Figure 7. The entity context manages the loading and saving of database objects under the general control of the configuration and entity context API. Usually, database updates are deferred until synchronization between the object cache and database is needed. When an object is accessed, the entity context provides the needed data out of the object cache, or if it is not found, the database is read. Thus rereads of database data by one thread are prevented (unless explicitly requested through the entity context API), preventing some repeated-read anomalies.

The database maintains a buffer cache of recently accessed rows, so the access to popular rows is very fast. This buffer sits logically in front of the disk data as is shared among all the apps, not just the ones using ORM. It is of course essential to provide enough memory in the database buffering system. This often requires changing the database configuration from the defaults of its installation. Database indexing is just as important as ever.

The entity context can handle a new id created on insert of a row for a new entity, even though id generation on insert is not available in standard SQL. Each database product has a way to do this, by an auto-incrementing datatype or "sequence". This useful SQL extension is made portable and attractive.

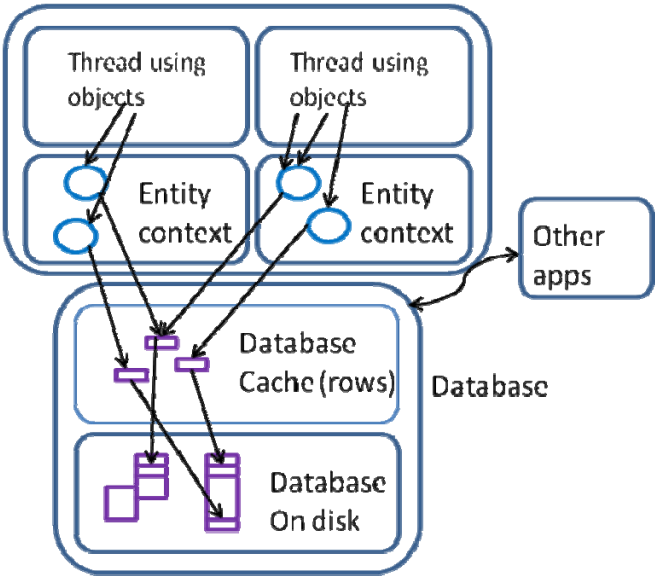

Figure 7. Entity contexts and corresponding database data

6. THE DOMAIN OBJECT LIFE CYCLE

When you create a new domain object, it has no connection to the entity context. Thus you must explicitly introduce it to the entity context with a Hibernate save or EDM AddObject operation. In EDM, every domain class is a subclass of EntityObject, a system class. In Hibernate, there is no such system superclass. On noting the system superclass for domain classes in EDM, you immediately worry about testability and code reuse, the bane of frameworks based on subclassing. However, the situation is not as bad as it might sound. C# provides "partial classes", so that all the system-provided code can be in one .cs file and all the application-provided code can be in another. The application-side partial class can be compiled by itself, wholly apart from the system classes, as part of a test program, for example.

In both systems, the data in the domain objects, obtained during the entity context lifetime, is still there after the entity context is closed down. In the typical application, the objects are filled out in the service layer and returned for display in the presentation layer *after the entity context is closed down*. After use in the presentation layer, the domain objects are discarded, having done their job delivering persistent data.

7. TRANSACTIONS

When the entity context actually interacts with the database, the database will start up a transaction if one is not yet running, and commit it after the current statement, a process known as auto-commit. To extend a transaction lifetime to contain multiple database accesses, an application needs to start up and commit a

transaction itself. These calls to start and finish a transaction are part of the entity context API. In a layered application, these calls are in the service layer, to allow the core business code to run the show. The commit of the transaction triggers a synchronization of the entity context to the database (in the simple use of the entity API).

There are three isolation levels to choose from: read committed (RC), read committed with versioning, or Serializable (SR). RC without versioning allows update anomalies without notification, and thus is not recommended. RC with versioning provides (very nearly) ANSI repeatable read (SR except allowing predicate anomalies) because of the rereads from the object cache. In versioning, the framework checks versions before writing and aborts the transaction if the database data has changed, as in snapshot isolation. However, note that full snapshot isolation stabilizes predicates where this does not. With the serializable level, versioning is redundant, accessed data is locked up, and deadlocks can occur, causing aborts. As usual, no free lunch.

7.1 Transactions vs. Entity Context Lifetimes: Are Contexts for Conversations a Good Idea?

Each transaction lives within a certain entity context. The simplest setup, used in the pizza project, has one transaction in each entity context lifetime. However, in general, an entity context can contain a sequence of transactions, possibly with large delays *between* the transactions, usually caused by UI actions. This multi-transaction scenario in an entity context is called a "conversation". In this case, in the times between transactions, the objects in entity context may stray from their "official" database values, as other activities change things. To help avoid such problems, the entity context can be refreshed from the database by explicit call(s) to the entity context API, or the entity context can be dropped and recreated, the simple way. Dropping and recreating an entity context is probably not much more expensive than a full refresh. In the case of web apps, the possibilities of extremely long waits between user requests argue for the simple context-per-transaction approach, to avoid wasting memory on idle conversations.

8. THEORY

The development of Hibernate has been entirely in the realm of practicing software engineers, and has gone remarkably unnoticed in the academic world. Because of the best software architects' strong belief in "clean" solutions, the results hang together as a beautiful system. The development of EDM has had input from the model-mapping community led by Phil Bernstein, and they have published papers [1, 9] explaining how the relationship between the entities and tables can be seen as views, and their updates as view maintenance. Clearly this analysis should be applied to the Hibernate system to bolster its foundations and check it out for related potential problems. Schema evolution is also relevant of course.

The Hibernate community often uses a simplified UML class diagram to express the entity model. EDM also has such a diagram. It would be interesting to examine how the entity model restricts the general UML class diagram. One of the troublesome cases is (in E-R terms) an N-N relationship with attributes.

9. SCALING UP

At the small-scale level, all the software can run on one system, which is very convenient for development. For production use, we are considering the case that one database server with plenty of resources can handle the transactional load of the applications using it. To offload the database server, the web application itself can run on application servers on the same fast local network as the database server. Similarly, to offload the application servers, the web servers can run on dedicated servers also on the local network, usually separated by a firewall switch. The switch ensures that static content's network traffic is localized to the outer network. Further, if there is a lot of static data (images, etc.), a content delivery network can be used to offload the web servers. Thus the upper end of the relevant range of applicability of this scenario involves many servers and can handle many simultaneous users.

Consider the case that each request that reaches the application server gets its own entity context. Then the database server's cache is providing the memory cache for all the domain data used by the application servers. The application servers themselves do not need to cache domain data except in the entity contexts themselves, and that is only for the current request cycle, each of which should last for less than one second. Although popular domain data is retrieved over and over from the database to the app servers, the retrieval is much faster than disk retrieval, since popular data stays in the database cache, and network transfer on a local network has latency in microseconds, compared to multiple milliseconds for disk access. Further, the total data transfer rate from database to app servers is easily seen to be trivial for today's fast local networks, as long as the domain objects themselves are not huge.

Figure 8 shows two concurrent requests being handled in one application server, each with its own entity context (object cache) getting data from the common database cache, or occasionally from the disk.

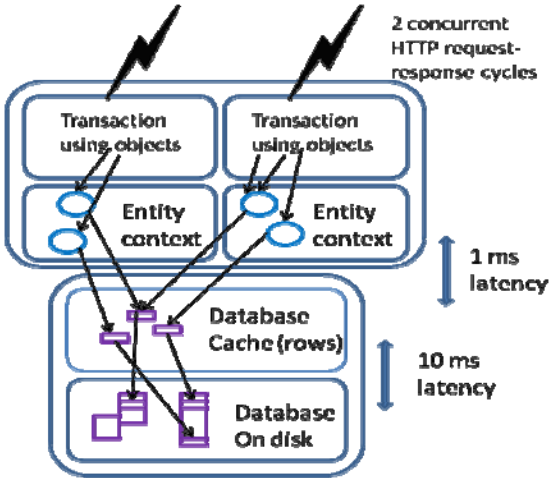

Figure 8. Data Access in a Web Application, where the database & application server reside on a fast local network.

If the application outgrows this simple one-database-server solution, it means the website is quite successful and should be able to afford investment in reengineering. If most of the domain data can be classified as application-specific slow-changing non-

critical data, a second-level cache can be provided in the application servers to offload the database for this kind of data, leaving the database to directly handle the core application-shared and/or money-related data. Above this size, we are considering truly large-scale sites, another topic.

10. THE FUTURE

The basic systems are in place for very useful designs, but it's still too hard to build them from scratch, or even from an existing database schema, except in the simplest cases. A programmer's workbench can and will be built, to suggest possible refactorings and help with incremental additions to a model, working from the Entity Model side or the object side. The competition between the Java community and the C#/Microsoft community will at least be interesting and may bring us more useful tools and other innovations.

About the presenter: Elizabeth (Betty) O'Neil is a Professor of Computer Science at the University of Massachusetts at Boston. She has worked with her husband, Patrick O'Neil, on research in database performance, indexing, and on authoring a database textbook. She has worked as a database system software developer for Sybase IQ, Microsoft, and Amdahl, and has developed and taught courses in database-backed web applications as well as database systems.

11. REFERENCES

[1] Adya, A, Blakely, J, Melnik, S, Meralidhar, S., and the ADO.NET Team, 2007, Anatomy of the ADO.NET Entity Framework, In Proceedings of SIGMOD 2007, ACM Press, New York, NY.

[2] Ambler, S., 2003. Agile Database Techniques, Wiley

[3] Bauer, C, King, G 2006 Java Persistence with Hibernate, Manning

[4] BEA 2006, Kodo Developer's Guide for JPA/JDO http://e-docs.bea.com/kodo/docs41/full/html/index.html

[5] Bernstein, P.A., Melnik, S. 2007 Model Management 2.0—Manipulating Richer Mappings. In Proceedings of SIGMOD 2007, ACM Press, New York, NY, 1-12.

[6] Chen, P. 1976 The Entity-Relationship Model—Toward a Unified View of Data, ACM Trans. on Database Systems, 1, 1, (March 1976), 9 – 36

[7] Hibernate, http://www.hibernate.org

[8] IBM, 2007 Feature Pack for EJB 3.0 for WebSphere Application Server V6.1http://www-1.ibm.com/support/docview.wss?rs=177&uid=swg21287579

[9] Melnik, S., Adya, A., Bernstein, P., 2007 Compiling Mappings to Bridge Applications and Databases, In Proceedings of SIGMOD 2007, ACM Press, New York, NY.

[10] MSDN Library, 2006 The ADO.NET Entity Framework Overview, http://msdn2.microsoft.com/en-us/library/aa697427(VS.80).aspx

[11] Oracle, 2008, Oracle TopLink 11g Preview, http://www.oracle.com/technology/products/ias/toplink/preview/index.html

[12] Unified Modeling Language. http://www.uml.org/