CocoBase® Enterprise O/R

Dynamic Mapping for the Enterprise Version 4.0

Object to Relational Mapping Concepts

US HEADQUARTERS / 657 Mission St. Suite 202
San Francisco, CA. 94105 / Tel.415-836-9199 /
Fax.415-836-9191

Object to Relational Mapping Concepts

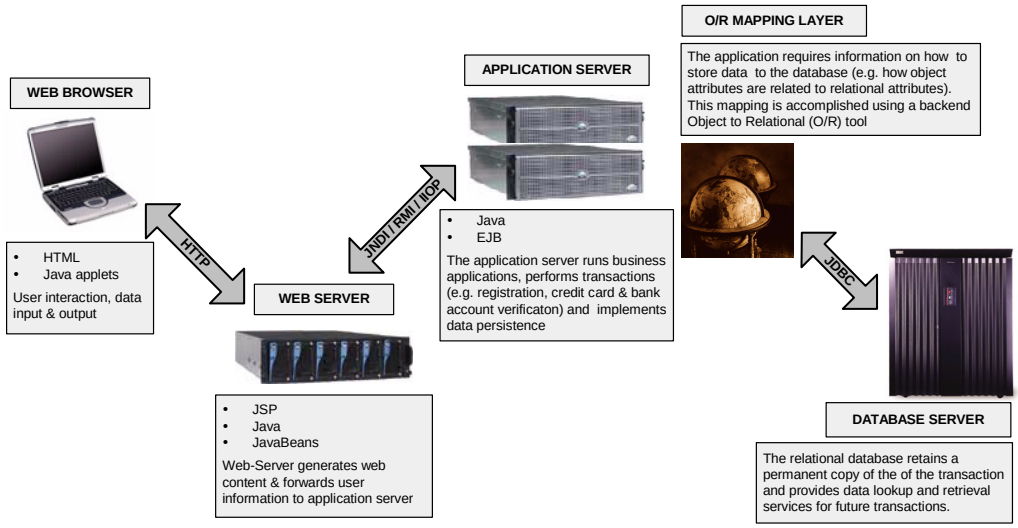
Document Number: CB-MC-R003-101702

Thought Incorporated.
657 Mission Street, Suite 202
San Francisco, CA 94105
USA

Printed in USA.

DataBase Mapping Systems for Java

In virtually all enterprise applications, there is a need for permanent storage of data related to a given business situation. An internet purchasing transaction is an example of such a situation, where customers log onto a website, register their name, address, telephone number, credit card number and other vital information, then proceed to purchase goods and services online. This information is likely to be saved or "persisted" to a database using a data mapping layer integrated into the host application.



This application, typically running in an application server, may have been written in one of a variety of programming languages and may implement several protocols for network services. Our focus is on Java and Enterprise JavaBeans (EJBs) and the associated protocols and APIs including CORBA, RMI, IIOP, JNDI and JDBC.

Do I Need O/R Mapping?

Data organization requirements for relational databases are often different than for the business applications written in object oriented languages. Relational database administrators tend to group closely related data such as *Street Number*, *Street Name*, *City*, *State*, and *Zip Code* into distinct data modules (e.g. an ADDRESS table), then link peripheral information through relationships. The ADDRESS table may be linked to a PERSON table containing the

person's *Last Name*, *First Name*, *Birthdate*, and *Social Security Number* as attributes. These attributes are generally immutable for each person. Several such links may exist for a given database table, as shown in *Figure 1*.

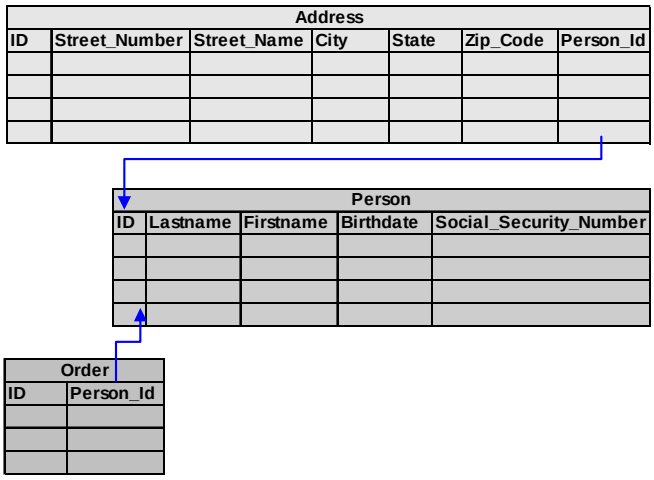


Figure 1 Table Relationships

Business applications may have different data requirements. For example, grouping *person* and *address* data into a single object may improve application performance by reducing the number of objects in memory, by allowing a single serialization call in a distributed application, or by reducing the number of database queries that are issued. Because relational data is not generally expressed in flat data models, but rather, grouped into modules that are connected through relationships, it is often difficult to normalize relational information into a flat object model. The structural differences between object and relational data representation is termed the *impedance mismatch*.

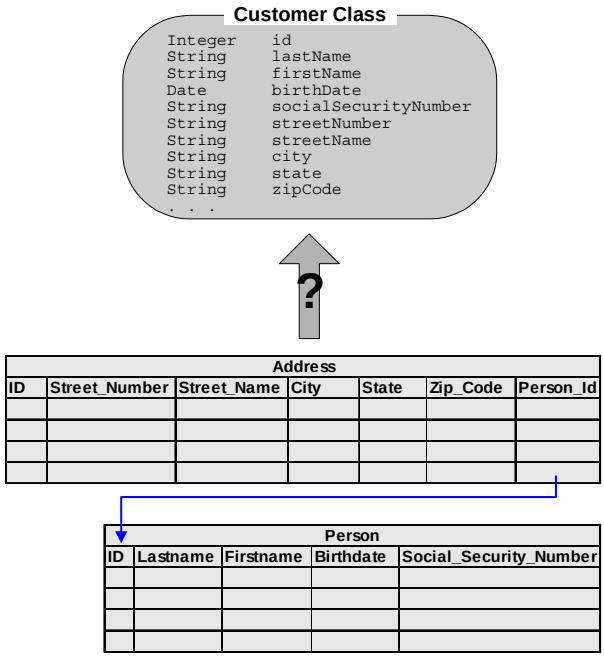


Figure 2 Impedance Mismatch

Several commercial object to relational (O/R) mapping tools are available to simplify the process of mapping relational data into the object space. The mapping strategies and implementations for these O/R tools are presented later in this document.

Java and EJB

Java

Java is an object oriented programming language that is similar in syntax to C/ C++ without pointer notation. Java was developed and is maintained by Sun Microsystems, and is designed to insulate applications from low level platform dependencies. This is done by creating a runtime platform that acts as a buffer between the application written in Java, and the underlying operating system and hardware. Applications written in Java are compiled into byte-code, which is a meta-state that can be efficiently interpreted and executed by the Java runtime facilities.

EJB

EJB is a specification, based on the Java programming language, for developing robust, scalable, enterprise level, server-side components. Among its major benefits are built-in transaction management, persistence and distributed components technologies. If a fault occurs during an EJB transaction, such as a banking account update, the entire transaction is rolled back to a known valid state. The transaction then proceeds, as before, with a data set that is known to be valid. EJB applications can interface with remote clients and databases. An EJB commerce application running in North America may have databases in Europe and a client application serving a customer base in Asia. The EJB architecture builds upon existing and mature technologies such as CORBA and SQL which have proven to be robust.

Because Java and EJB architectural specifications are actively maintained by Sun Microsystems and the Java community, and have the support of a countless number corporations, these technologies have gained huge momentum in the enterprise development arena.

Mechanisms of EJB

The two basic EJBs types are referred to as session beans and entity beans. Session beans generally represent a process and are used to manage that process, while entity beans represent specific, persistent, database oriented objects.

While both entity beans and (stateful) session beans can retrieve data from a database, typically only entity beans persist data to the database. If a *shopping cart* session bean manages an online e-commerce transaction, generally it would do so by delegating specific

tasks to subordinate entity beans such as an *order* bean or *customer* bean. These entity beans would be responsible for persisting the transaction data which may include the type and quantity of items purchased and customer address and order number.

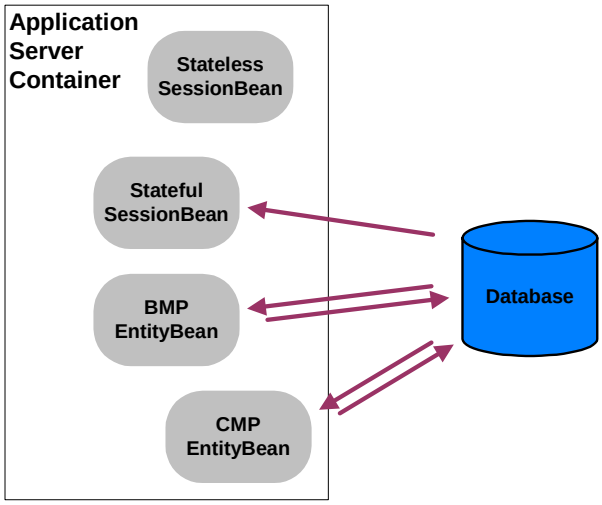


Figure 3 Enterprise JavaBeans

An EJB executing a purchasing transaction will eventually need to make a permanent record of the transaction if the host merchant is to be profitable. Surely, if the transaction is purely transient, and the company loses all information regarding the transaction once it is completed, the company will be short lived. To make a permanent record of the transaction, the EJB persists its data to the database.

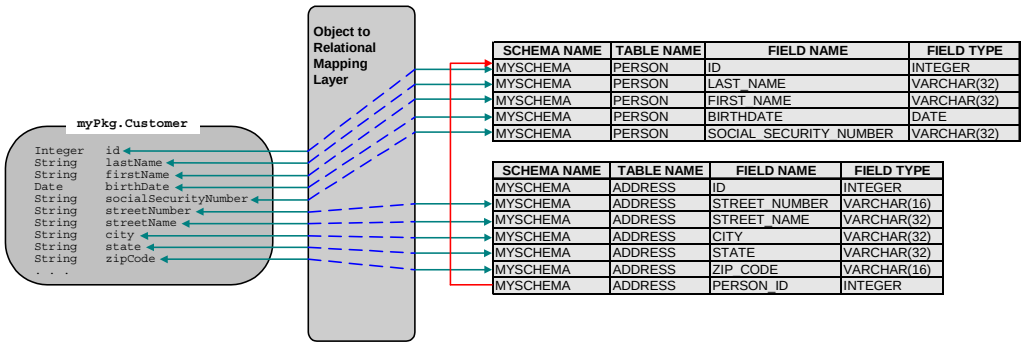


Figure 4 Object to Relational Mapping

Data persistence creates a permanent copy of the result set of some transaction or process that has occurred in the application. This data can then be retrieved for further processing or for reference at some later date. In *Figure 4*, attributes from an EJB representing a customer are updated to a pair of database tables. Each EJB attribute corresponds to at least one table attribute. To create this relationship, a mapping scheme must exist to pair the object and table entries. The mapping becomes more complex when attribute mapping spans multiple database tables.

Enterprise data persistence is accomplished through one of four industry accepted mapping techniques. Each of these techniques is presented in detail below:

The Hand Coded or Embedded JDBC Method

The first and most common persistence mechanism is embedded JDBC API calls in the application source code. With this method, a specific JDBC driver implementation is statically coded into the persistent object or application source code. The API calls are executed when a persistence facility is required.

One of the major advantages of embedding persistence code directly in the object is that a developer may already be familiar with the development process for a particular database or JDBC driver. The developer may also be familiar with the specific optimizations for a particular application as well. Because JDBC driver implementations are readily available, they have become the API that many developers rely on for their persistence requirements. Database vendors often prefer this method of persistence because it is easy to support and optimize applications implementing their proprietary JDBC implementations.

The main disadvantage of using embedded JDBC calls is that customers are usually locked into one vendor and one database, making the prospect of writing portable components or of porting components to a different database a daunting task. If the developer needs to update, optimize or port an application, the persistence functionality must often be de-coupled from the application code, re-coded to the new API, then debugged.

Tight Object-Map Binding

The second common persistence mechanism uses a tightly bound persistence layer. This mechanism is supported by the ODMG and is used by some common object to relational (O/R) tools. This technique binds an object model to a specific mapping definition by embedding JDBC interface calls into the mapping layer or application byte code. This, in effect, creates an inflexible situation similar to the hand coded mechanism.

The main advantage of this mechanism is that it provides a maintenance benefit over hand-coded JDBC implementations. If the object model changes or the application must be ported to another database, the O/R vendor provides tools to regenerate the persistence layer for the new target.

Updating and extending the object model, the application, or mapping definitions may be conceptually simple, but is often tedious, with most tools requiring regeneration of the source code, re-compilation, then re-deployment of the application if persistence requirements change.

When byte code manipulation is implemented by the mapping tool (as is the case with the ODMG model), debugging the application can become a frustrating and time consuming task. With this method of mapping, the risk of EJBs or other standardized object models becoming non-comformant or incompatible with new or emerging standards increases drastically.

If a tightly bound map implementation is used with an optimized EJB or a generic Java data object, in order to access the same database attributes, each object model must be individually mapped and have multiple mapping layer support points to achieve the same mapping scheme.

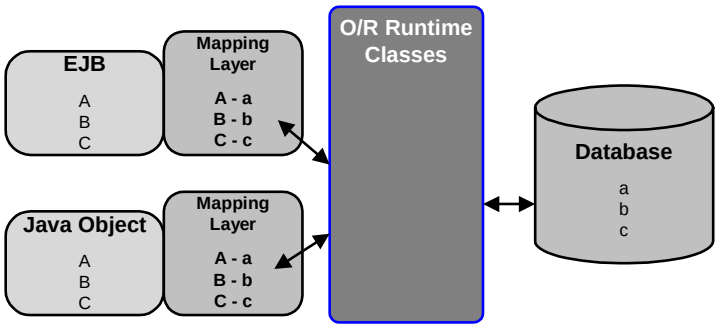


Figure 5 Tightly Bound Mapping Layer

To implement a simple change in the application, such as adding an attribute to the database table, or adjusting the generated SQL, each affected mapping definition for each object model must be updated individually. The ODMG and the embedded mapping techniques tend to break down for the enterprise because they are too tightly bound and inflexible and don't offer an easy way to enhance the functionality or mapping behavior an application. Additionally, the associated mapping tools don't provide much extensibility due to the nature of the statically bound, one class - one map model.

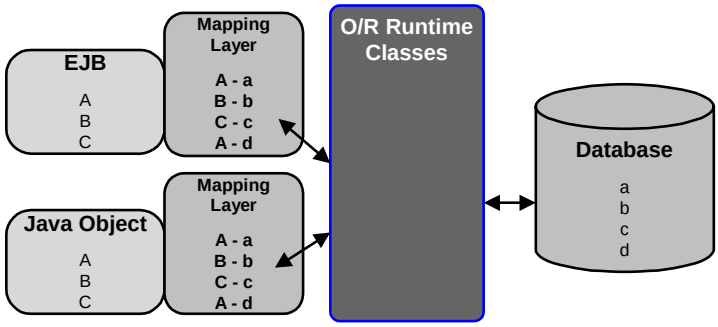


Figure 6 Redefined Map Definition - Tightly Bound Mapping Layer

Many developers are initially drawn to tight binding technique because the O/R vendor claims that this approach is transparent to the database or uses simple post processing tools. This is also a more familiar method for map implementations because of its relative lack of sophistication leading to a simpler object structure. While the one class - one map approach may seem attractive at first glance, once the developer enters the maintenance cycle or initiates a moderate to large project, this rigid architecture represents a substantial maintenance or cost commitment due to the static, black box nature of this approach.

Container Managed Persistence

The third persistence mechanism is container managed persistence (CMP). This persistence mechanism is provided for EJB object models only. With CMP, only data attributes and business methods are implemented in the source code. Persistence layer calls are unnecessary because persistence is automatically provided for any exposed attributes by the application server container, once the EJB is installed.

The main advantage to using the CMP facilities provided by the application server is that persistence is automatic. Once a JDBC datasource is configured for the server, the developer does not require any knowledge of the JDBC API to access the JDBC layer. Additionally, applications can take advantage of any caching facilities that may be provided by server.

In general, application servers do not provide O/R mapping capability. Object models utilizing application server CMP tools must be a simple projection of the database table. If the database data structure includes several foreign key relationships, data cannot be normalized (i.e. flattened) in the object space.

Application server CMP tools do not provide a persistence solution for other enterprise object models such as BMP and Java objects. This is a risky proposition for an enterprise application that requires a higher level of scalability and extensiblity.

Application server CMP tools often perform poorly in comparison to commercial O/R tools with CMP support. EJB query protocol specifies that a collection of EJB primary key objects is first retrieved then used to perform individual remote interface lookups for the EJB as required. A CMP implementation that doesn't provide substantial resource optimizations can very quickly become useless for enterprise level applications.

Dynamic Object to Relational Mapping

The fourth persistence mechanism, innovated by Thought Incorporated, is the dynamic, highly scalable mapping model. With this model, database maps can be shared across object models. Servlet, JSPs, applets, EJBs, and Java objects, can simultaneously share the exact same mapping definition. With the CocoBase architecture, mapping definitions are loaded dynamically into the mapping layer.

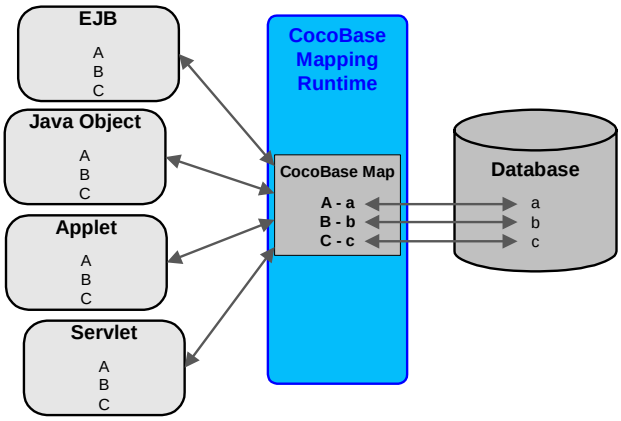


Figure 7 Heterogeneous Environment Support with CocoBase

To performance tune or port an application or to change or optimize a database, a single map definition can changed and that change is reflected across all objects using that map. This becomes a huge benefit in real enterprise application development where many objects models may share a common mapping definition.

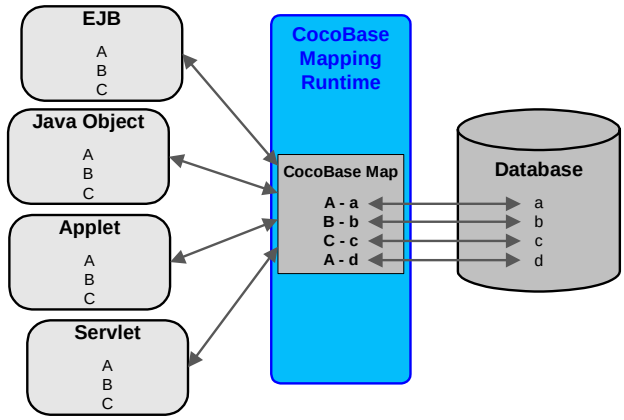


Figure 8 Reconfiguring the CocoBase Map

Using this mapping technique, an IT department can create application components independently from the Sales and Marketing or the Accounting departments yet share the same database and database map for their distinct object models. The Sales and Marketing department can submit their quarterly sales figures to a central database using a Java based application with dynamic mapping support. These figures can then be retrieved by the Accounting department using an EJB based application and the same persistence maps.

The CocoBase tools also generates non-proprietary Java code elements for an variety of object models. Generic Java objects, JSPs, and EJB components can generated as source code for supported platforms, and can optionally incorporate CocoBase mapping runtime calls. All code generation templates are exposed and user extensible.

CMP support is provided through the optional CocoBase CMP installer. Using the CMP installer, EJBs can take advantage of the dynamic mapping capabilities provided through the CocoBase runtime. CMPs are not limited to simple table projections as with application server CMP support. Additionally, with the CocoBase CMP installer, CMP implementations can be targeted, on the fly, for any supported application/EJB server. Persistence methods are added to the original implementation through subclassing, preserving the original generic classes. The resulting persistent subclasses can be immediately installed on the server and are also provided as source code.

Another advantage of the CocoBase dynamic model is that custom behaviors, such as caching or custom data routing, can easily be implemented, on the fly, through the plug-in API. For example, if the central database is found to perform poorly, an in-memory cache database can be installed on the fly, without changing the object or mapping models. When a plug-in cache is registered with the CocoBase runtime, it is first queried for the specified

data. If the data is not located in the cache, a query is issued against the central database. CocoBase allows caching to be supported across distributed object models and for different server and database vendors.

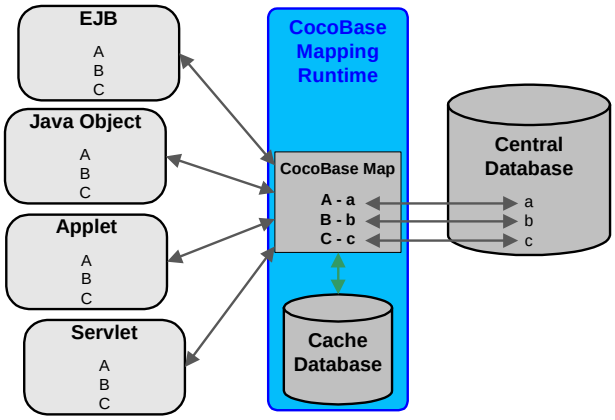


Figure 9 Plug-in Caching with CocoBase

By contrast, the hand-coded and tightly bound mapping solutions require manual changes to each driver hook in each object, in order take advantage of a new caching system. Each object would then require re-compilation and redeployment to implement this new functionality.

With the dynamic mapping approach applications are made faster, more scalable and more extensible than with a static, embedded persistence approach. Developers often believe that a static model will perform more efficiently than a dynamic model because there's little or no mapping layer. The CocoBase mapping layer is highly optimized and the integrated enhancements are so sophisticated that applications using the CocoBase mapping run much faster than hand written or statically bound O/R-centric applications. In some cases, up to 4000% faster.

These benefits are achieved without requiring object inheritance from special classes or implementing proprietary interfaces. CocoBase Enterprise provides the most widely accepted standard in the industry for O/ R mapping, without requiring custom or proprietary changes to Java classes. If this technology is coupled with database gateways, multiple databases can be updated simultaneously with a single method call. This is just one example of the flexibility afforded developers using CocoBase Enterprise. This kind of enhancement would be prohibitively expensive to implement using the other, less sophisticated mapping approaches.