Enterprise Objects Framework Developer's Guide

This manual describes EOF 3.0.

Writing: Kelly Toshach
With help from: Bruce Arthur, Nancy Craighill, Craig Federighi, Patrick Gates, Stan Jirman, Kenny Leung, Mai Nguyen, Eric Noyau, Steve Miner, and Greg Wilson
Design and Illustration: Karin Stroud
Production: Gerri Gray

Contents

Table of Contents

Introduction

Enterprise Objects Framework brings the benefits of object-oriented programming to database application development. You can use the Framework to build feature-rich, database applications with reusable software components that tightly couple business information with the business processes managing that information.

One of the most significant problems developers face when using object-oriented programming languages with SQL databases is the difficulty of matching static, two-dimensional data structures with the extensive flexibility afforded by objects. The features of object-oriented programming—such as encapsulation and polymorphism—and their benefits—like fewer lines of code and greater code reusability—are often negated by the programming restrictions that come with accessing SQL databases within an object-oriented application.

Enterprise Objects Framework solves this problem by providing tools for defining an object model and mapping it to a data model. This allows you to create objects that encapsulate both data and the methods for operating on that data, while taking advantage of the data access services provided by the Framework that make it possible for these objects to persist in a relational database.

The flexible, three-tier architecture provided by the Framework allows you to build robust, scalable, client/server applications. Objects at each of the three tiers (user interface, enterprise objects, and data store) can be deployed to take advantage of network resources. For example, data might be stored in a relational database running on a fault-tolerant database server with gigabytes of disk storage, while enterprise objects run on high-end compute servers. Partitioning the application to make best use of available resources allows complex applications to achieve maximum performance.

The components of Enterprise Objects Framework fully embrace the three-tier architecture, which means that portions of the Framework can be used selectively to meet specific application requirements. For example, the components that provide users with the ability to interactively manipulate enterprise objects can be used by a non-database application to handle user interface refresh and undo. You can use a custom data store (such as a flat-file system) in place of a relational database to store data for enterprise objects. Or you can

make use of the database adaptors separate from the rest of the Framework components to provide direct access to relational databases for your applications.

Enterprise Objects Framework offers these additional benefits:

- *Flexibility.* An enterprise object isn't constrained by the physical location of data. Its mapping can extend across tables, and its data isn't confined to the object's mapping to a physical database. Further, the mapping of an enterprise object to the database can be dynamically controlled at run time.
- *Modularity.* Depending on the needs of your application, you can create simple applications that require little or no code, program selected components while accepting the default behavior of other components, or use selected components independent of the rest of the Framework.
- *Extensibility.* Enterprise Objects Framework's classes are public and extensible. For example, you can provide your own data source, or add support for a new user interface object.

About This Book

This book describes concepts that you'll need to know when writing a Enterprise Objects Framework application. To help you find what you are looking for, this book is organized into three parts:

- Part 1, "Enterprise Objects Frameworks Essentials" on page 17, provides an overview of how Enterprise Objects Framework works and the different types of applications you can build with it. The first chapter, "What Is Enterprise Objects Framework?" on page 19, describes what Enterprise Objects Framework is, how it's different from other products, and what features it offers. The second chapter, "Enterprise Objects Framework Viewed Through Its Classes" on page 43, provides a description of the classes used in Enterprise Objects Framework applications and how they fit into different types of applications.

- Part 2, "Enterprise Object Design" on page 63, describes how to implement business logic for your application. With Enterprise Objects Framework, you put business logic in business objects, called *enterprise objects*. The two chapters in Part 2, "Designing Enterprise Objects" on page 65 and "Advanced Enterprise Object Modeling" on page 105, describe the tasks you perform in defining the enterprise objects for your application.

- Part 3, "Application Design" on page 139, describes how to implement application-level logic for your application. Enterprise Objects Framework provides a basic structure for all applications that use it. The chapters in this section— "Application Configurations" on page 141, "Connecting to a Database" on page 173, "Behind the Scenes" on page 187, and "Answers to Common Design Questions" on page 221—describe that structure, how it's established at runtime, and how to intervene in its default behaviors.

There are no prerequisites for learning Enterprise Objects Framework; however, it does help if you understand something about relational databases and Entity-Relationship modeling. If you aren't familiar with these topics, read the Appendix, "Entity-Relationship Modeling" on page 257. While Enterprise Objects Framework largely encapsulates the programmer from having to know about relational databases, you still need to understand Entity-Relationship Modeling to map your enterprise objects into a relational database for storage. Entity-Relationship Modeling terminology is used by the Enterprise Objects Framework classes and documentation to describe this mapping.

Part I

Enterprise Objects Frameworks Essentials

Chapter 1

What Is Enterprise Objects Framework?

Enterprise Objects Framework is a set of tools and resources that help you create applications that work with the most popular relational databases—or with your own custom data store. These tools don't help you build a complete database system from the ground up—the tasks of data storage and retrieval are left to a database server supplied by a third party. Rather, Enterprise Objects Framework lets you design *object-oriented* database applications that are easy to build and maintain and that draw upon standard user interface features.

This chapter describes how Enterprise Objects Framework is different from other database access products, the features it offers, and how you can use it. The chapter is divided into the following sections:

The Enterprise Objects Framework Difference

Today's business applications must embody complex rules of the business, access heterogeneous corporate data in database systems from multiple vendors, and offer different front ends to meet the needs of users in all different parts of the business. This is a tall order to fill, but Enterprise Objects Framework meets all these needs. It provides database independence, transparently maps custom business objects to database tables, and binds business objects to user interfaces.

Where Does Business Logic Go?

The biggest difference between Enterprise Objects Framework and other solutions is where you put business logic. One approach is to implement business rules in the user interface, as you do with 4GL tools. Problems with this approach include:

- *It offers limited reuse.* You have to code your business logic into each application that accesses your database. In fact, within an application, you have to code your business logic into each screen. Consequently, you wind up duplicating your code.
- *It's not maintainable.* Since you have to duplicate your business logic, even small modifications to your rules are difficult to implement. Finding and fixing every affected screen in every affected application is slow and error prone. Modifications to your database schema are equally problematic.
- *Different user interfaces require different implementations.* For example, if you have a client/server application that you want to put on the web, you have to rewrite the application and maintain both versions.
- *It provides poor data integrity.* You have to rely on all application developers to implement the business rules correctly. If any screen of any application has an error, the data in your database can be corrupted, impacting all applications.
- *It doesn't scale well.* To improve your application's performance, you have to provide your users with faster systems. Contrast this with a solution in which you can move some computation-intensive processing to fast server machines.

Another approach is to implement your business rules in the database—with stored procedures, rules, constraints, and triggers, for example. This approach also has problems:

- *It offers limited interactivity.* To provide immediate feedback to a user, you have to make a round trip to the database every time the user performs an action, which can be very slow and inefficient. On the other hand, you can batch up changes, but then the user doesn't receive immediate feedback.

- *No back-end portability.* Database vendors all have different ways to implement logic. If you have to support more than one database, you'll have to implement the logic multiple times, resulting in more maintenance problems.

- *SQL is a poor development language.*

A third approach—the one that Enterprise Objects Framework takes—is to put business rules in business objects, called *enterprise objects*. By applying good object-oriented design principles, this approach provides the advantages of encapsulation, reuse, and a more natural model of the real world. For example, suppose you're writing an application for managing a video rental store. The business logic for such an application might include the rules:

- A late fee is generated automatically when a rental becomes overdue.
- A customer can't rent more videos if they have any overdue rentals.
- The total replacement value of a customer's rentals can't exceed the amount of the customer's deposit.

With Enterprise Objects Framework, you would implement these rules in enterprise objects such as Customer, VideoTape, Rental, and Fee.

What Doesn't Go in an Enterprise Object

Deciding what code to leave out of your business objects is just as important as deciding what code you leave in. To maximize the reusability and extensibility of your objects, they shouldn't embed knowledge of the user interface or database alongside the business logic. For example, if you embed knowledge of your user interface, you can't reuse the objects because each application's user interface is different; and if you embed knowledge of your database, you have to update your objects every time you modify the database.

If not in the business objects, then where does this knowledge go? It's handled by Enterprise Objects Framework as shown in Figure 1.

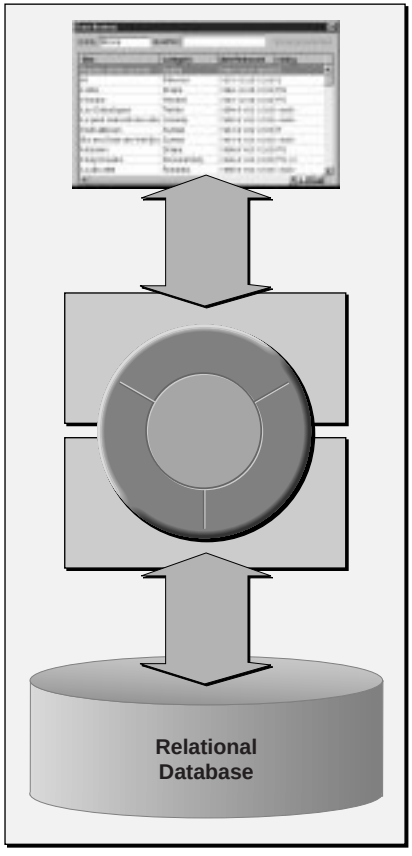


Figure 1. The Enterprise Objects Framework Approach

The Framework provides a database-to-objects mapping so your objects are encapsulated from the database, and it provides an objects-to-interface mapping so they are encapsulated from the user interface. This approach enables you to create libraries of enterprise objects that can be used in as many applications as you need, with any user interface, and with any database server. You're able to concentrate on coding the logic of your business while the Framework takes care of the rest.

From Database to Objects

Enterprise Objects Framework's database-to-objects mapping sets up a correspondence between database tables and enterprise objects classes so that database rows map to instances of the appropriate class as shown in Figure 2.

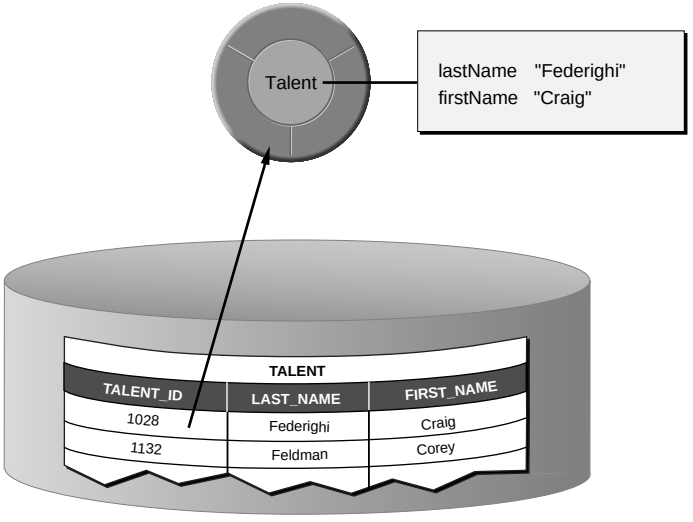


Figure 2. Mapping Between an Enterprise Object Class and a Single Table

The mapping is flexible. For example:

- You can map an enterprise object to a single table, a subset of a table, or to more than one table. For instance, a Person object can get its first and last names from a PERSON table but get its street address, city, state and zip code from an ADDRESS table.
- Generally an enterprise object instance variable maps to a single column, but the column-to-instance variable correspondence is similarly flexible. You can map an instance variable to a derived column, such as "price * discount" or "salary * 12".
- You can map an enterprise object inheritance hierarchy to one or more database tables.

In addition to mapping tables to enterprise object classes and database columns to instance variables, the Framework maps database primary and foreign keys to relationships between objects. The Framework defines two types of relationships—to-ones and to-manys—which are both illustrated in Figure 3. The relationship a MovieRole has to its Movie is a to-one relationship, while the relationship a Movie has to its MovieRoles is a to-many.

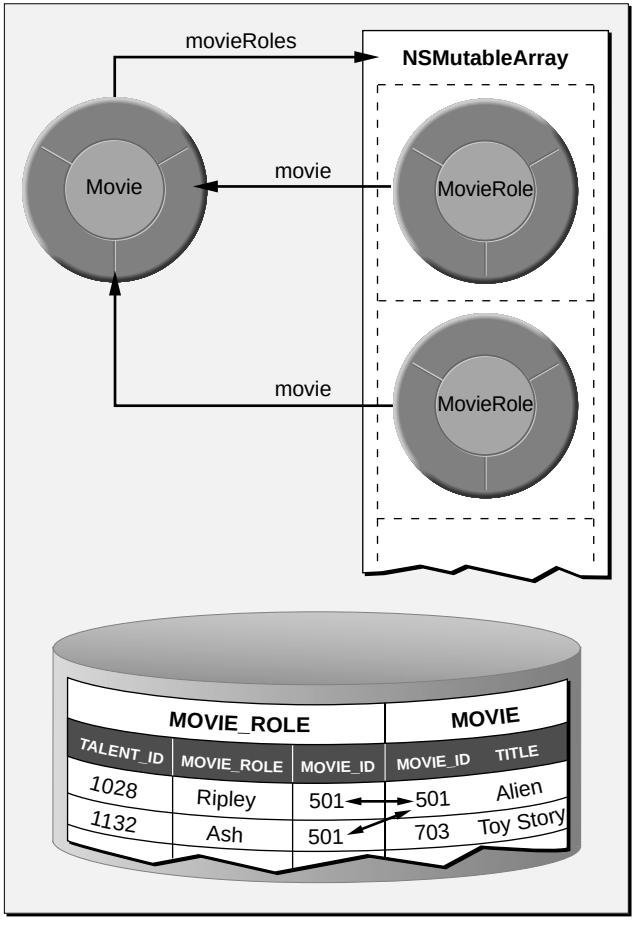


Figure 3. Mapping Relationships

For more information on database-to-objects mappings, see the chapter "Designing Enterprise Objects" on page 65, and to learn how to define this mapping with the EOModeler application, see the book *Enterprise Objects Framework Tools and Techniques*.

Uniquing

In marrying relational databases to object-oriented programming, one of the key requirements is that a row in the database be associated with only one enterprise object in a given context in your application. Enterprise Objects Framework maintains the mapping of each enterprise object to its corresponding database row, and uses this information to ensure that your object graph does not have two (possibly inconsistent) objects for the same database row. *Uniquing* of enterprise objects, as this process is called, limits memory usage and allows you to know with confidence that the object you're interacting with represents the true state of its associated row as it was last fetched into the object graph.

Without uniquing, you'd get a new enterprise object every time you fetch its corresponding row, whether explicitly or through resolution of relationships. This is illustrated in Figure 4.

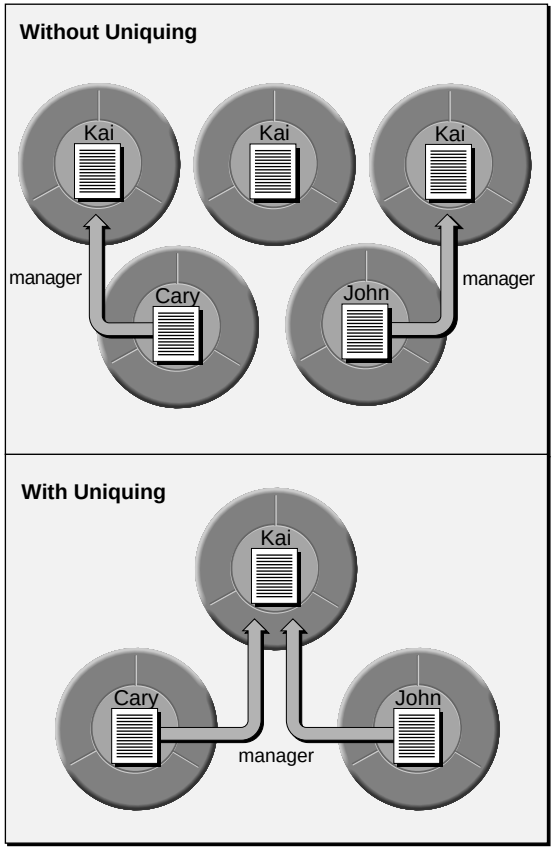


Figure 4. Uniquing of Enterprise Objects

This shows the enterprise objects that would exist after fetching three employee objects without uniquing. Kai is Cary's and John's manager. On fetching an object for Cary, an object representing Kai is created to resolve the manager relationship. If you then fetch an object for Kai, a separate object is created. Fetching an object for John then causes yet another object representing Kai to be created. Kai's row in the database can be altered between any of these individual fetches, resulting in objects representing the same row, but with different data.

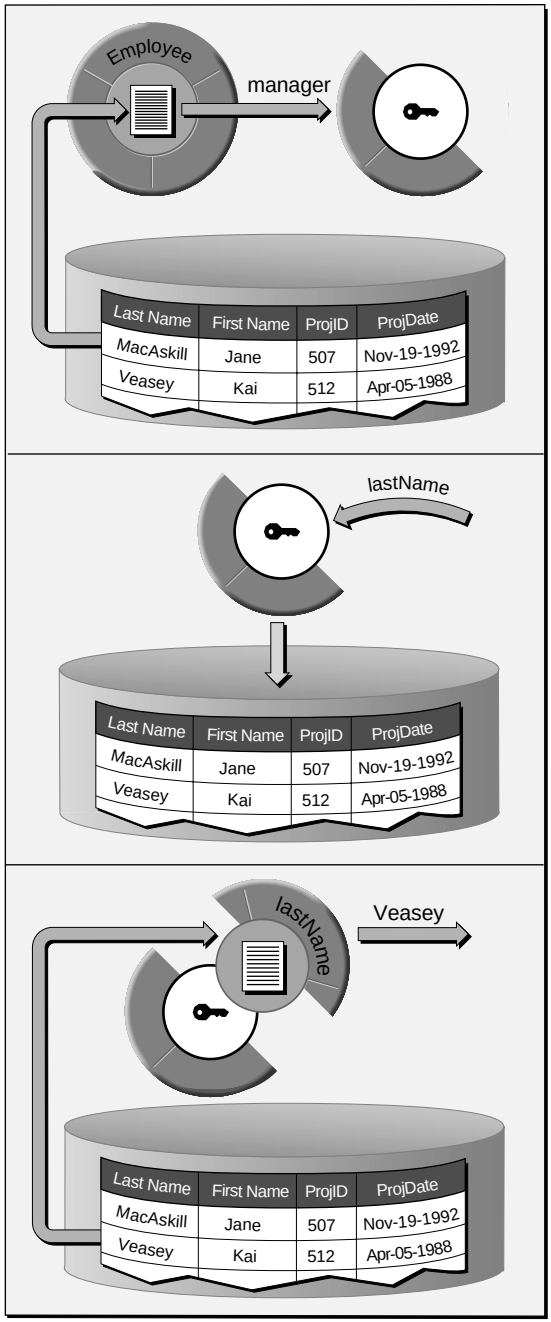
Using uniquing results in only one object ever being created for Kai. In this case, even though Kai's row can be changed, your application has a single view of Kai's data. The data may not reflect what's in the database if another user changes it, but there's no ambiguity within your application.

Resolution of Relationships and Faulting

When the Framework fetches an object, it creates objects representing the destinations of the fetched object's relationships. For example, if you fetch an employee object, you can ask for its manager and immediately receive an object; you don't have to get the manager's employee ID from the object you just fetched and fetch the manager yourself.

The Framework doesn't immediately fetch data for the destination objects of relationships, however. Fetching is fairly expensive, and further, if the Framework fetched objects related to the one explicitly asked for, it would also have to fetch the objects related to those, and so on, until all of the interrelated rows in the database had been retrieved. To avoid this waste of time and resources, the destination objects created are stand-ins, called *faults*, that fetch their data the first time they're accessed. Figure 5 illustrates this process.

The framework allows you to tune relationship resolution by *prefetching* relationships and *batch faulting*. For more information on these features, see the chapter "Answers to Common Design Questions" on page 221. For more information on the general faulting mechanism, see the chapter "Behind the Scenes" on page 187.



The Employee Object "Jane MacAskill" is fetched from the database. Instead of fetching the data for Jane's manager (Kai Veasey) right away, the Framework creates a fault containing the value of the foreign key for Jane's manager relationship. The graphic for the fault has an empty center with a key in it, indicating that it contains no real values yet. The bottom half of the object shows the messages the fault can respond to without first having to fetch its data.

The fault receives a message it can't cover for (lastName).

The fault fetches its data from the database and invokes its lastName method.

The string "Veasey" is returned.

Figure 5. Resolution of a Fault

From Objects to Interface

The objects-to-interface mapping takes care of automatically synchronizing the user interface with your enterprise objects. If a user changes a value in the user interface, the Framework updates the corresponding enterprise objects. Similarly, if an enterprise object is changed programmatically, the Framework automatically updates the user interface.

From Objects to Database

After your program has accumulated changes to enterprise objects, it needs to push those changes back to the database. Enterprise Objects Framework manages this process, too, analyzing the objects for changes, generating corresponding database operations, and executing those operations to synchronize the database with in-memory enterprise objects. The Framework has mechanisms for ensuring that the integrity of your data is maintained between your application and the database:

- Validation
- Referential integrity enforcement
- Automatic primary and foreign key generation
- Transaction management
- Locking

Each of these is described in the following sections.

Validation

A good part of your application's business logic is usually validation—for example, verifying that customers don't exceed their credit limits, that return dates don't come before their corresponding check out dates, and so on. In your enterprise object classes, you implement methods that check for invalid data, and the framework automatically invokes them before saving anything to the database.

Referential Integrity Enforcement

Enterprise Objects Framework allows you to specify rules governing the relationships between objects. You can specify whether a to-one relationship is optional or mandatory. For example, you can require that all departments have a location (mandatory), but not require that every employee have a manager (optional).

You can also specify delete rules for relationships. For example, when you delete a department object, you can specify that:

- All the employees in that department are also deleted (a cascading delete).
- All the employees in that department are updated to have no department (nullify).
- The department deletion is rejected if it has any employees (deny).

For more information on Framework's referential integrity enforcement, see the chapter "Designing Enterprise Objects" on page 65. To learn how to define these rules in the EOModeler application, see the book *Enterprise Objects Framework Tools and Techniques*.

Automatic Primary and Foreign Key Generation

With Enterprise Objects Framework, you don't have to maintain database artifacts such as database primary and foreign key values into your application. Database primary and foreign keys aren't usually meaningful parts of a business model; rather, they're attributes created in a relational database to express relationships between entities. For example, the primary key (MOVIE_ID) for a movie doesn't have any meaning to users. Users identify movies by their titles.

Enterprise Objects Framework keeps track of primary and foreign key data for you. You don't have to represent that information in your enterprise objects, and you don't have to worry about generating and propagating key values.

For information on eliminating database artifacts from your object model, see the chapter "Designing Enterprise Objects" on page 65. For information on how the Framework generates primary key values, see the chapter "Answers to Common Design Questions" on page 221.

Transaction Management

For the most part, Enterprise Objects Framework handles transactions for you. You don't have to worry about beginning, committing, or rolling back transactions unless you want to fine-tune transaction management behavior. The Framework uses the native transaction management features of your database to group database operations that correspond to the changes that have been made to enterprise objects in memory. For more information, see the chapter "Behind the Scenes" on page 187.

Additionally, the Framework provides a separate in-memory transaction management feature. You can create nested contexts in which a child context's changes are folded into the parent context only upon successful completion of an in-memory operation. For more information on nested contexts, see the chapter "Application Configurations" on page 141.

Locking

The Framework offers three types of locking:

- *Pessimistic*. With this strategy, Enterprise Objects Framework uses your database server's native locking mechanism to lock rows as they're fetched into your application. If you try to fetch an object that someone else has already fetched, the operation will fail because the corresponding database row is locked. This approach prevents update conflicts by never allowing two users to look at the same object at the same time.

- *Optimistic*. With this strategy, update conflicts aren't detected until you try to save an object's changes to the database. At this point, the Framework checks the database row to see if it's changed since your object was fetched. If the row has been changed, it aborts the save operation.

  Enterprise Objects Framework determines that a database row has changed since its corresponding object was fetched using a technique called *snapshotting*. When the Framework fetches an object from the database, it records a *snapshot* of the state of the corresponding database row. When changes to an object are saved

to the database, the snapshot is compared with the corresponding database row to ensure that the row data hasn't changed since the object was last fetched. For more information on snapshots, see the EODatabaseContext class specification in the *Enterprise Objects Framework Reference*.

- *On-Demand*. This approach is a mixture of the pessimistic and optimistic strategies. With on-demand locking, you lock an object after you fetch it but before you attempt to modify it. When you try to get a lock on the object, it can fail for one of two reasons: the corresponding database row has changed since you fetched the object (optimistic locking), or because someone else already has a lock on the row (pessimistic locking).

For more information on Enterprise Objects Framework's locking strategies, see the chapter "Behind the Scenes" on page 187.

Ingredients of an Enterprise Objects Framework Application

Enterprise Objects Framework can be used to create many different kinds of applications:

- Command-line programs without a graphical user interface
- Application Kit client/server desktop applications where all the logic is in a client
- HTML WebObjects applications where all the logic is in the application server
- WebObjects applications with interactive Java clients where the logic is distributed between an application server and its clients

Regardless of the type of application you're building (and assuming your data store is a relational database), creating an Enterprise Objects Framework application usually involves the following components (shown in Figure 6).

- *A user interface.* The type of interface you want—whether a graphical user interface based on Apple's Application Kit for a desktop application, an HTML web interface based on Apple's WebObjects framework, or an interactive Java client (using Sun's JDK user interface objects) for a web application—determines the tools you use to create the user interface. For Application Kit applications and Java web clients, you use Interface Builder. For HTML web applications, you use WebObjects Builder.

- *A model.* A model defines the mapping between your enterprise objects and the data stored in your database. An enterprise object class typically corresponds to a table in a database, and an enterprise object instance corresponds to a single row or record in the corresponding table. You define and store this correspondence in models that you build graphically with the EOModeler application.

- *Enterprise Objects.* These are your business objects, in custom code that you provide. Enterprise objects couple data from the database with the business logic required to operate on that data.

- *Enterprise Objects Framework's classes and interfaces.* The classes and interfaces (or classes and *protocols* if you're using Objective-C) let you programmatically manipulate data as it passes between the database server, your enterprise objects, and the user interface. Although simple applications can be created entirely in one of the builder tools, sophisticated applications require using some of the Enterprise Objects Framework classes in your own code.

- *A database server and an adaptor for that server.* An adaptor is a mechanism that connects your application to a particular server. For each type of server you use, you need a separate adaptor. Enterprise Objects Framework provides adaptors for Oracle, Sybase, Informix, and ODBC-compliant servers. It also provides a sample adaptor for a flat-file data store and an adaptor for OpenBase Lite—a database that ships with Enterprise Objects Framework as an unsupported demo.

What varies between different types of applications is the parts of the Framework that you use and how they interact with your application's user interface.

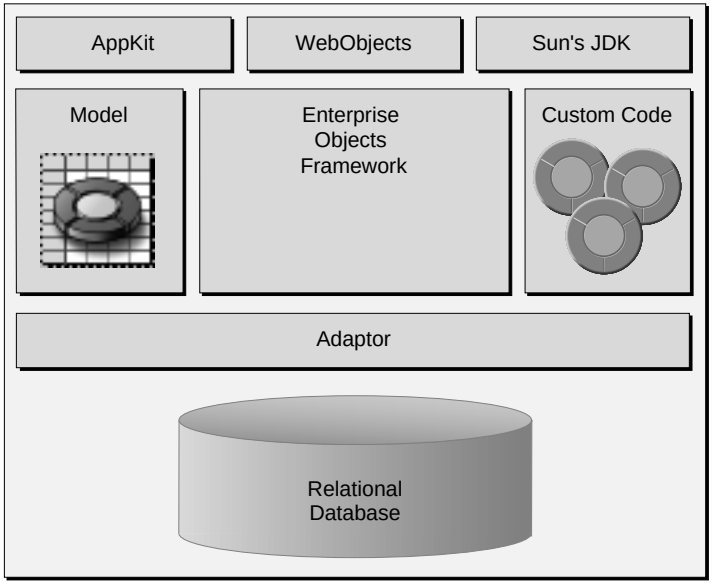


Figure 6. The Ingredients of an Enterprise Objects Framework Application

Enterprise Objects Framework Layers

Conceptually, Enterprise Objects Framework is divided into four layers: the access layer, the control layer, the interface layer, and the distribution layer. Correspondingly, it partitions its classes and interfaces into four frameworks: EOAccess, EOControl, EOInterface, and EODistribution. The names of the layers and frameworks are used interchangeably throughout the Framework documentation.

This section introduces the Framework layers by demonstrating the roles they play in the four most basic types of Enterprise Objects Framework applications.

A Command-Line Program

The simplest type of application is a command-line program (any program that doesn't have a graphical user interface). As shown in Figure 7, a command line program uses only the most fundamental layers of the Framework: the access layer and the control layer.

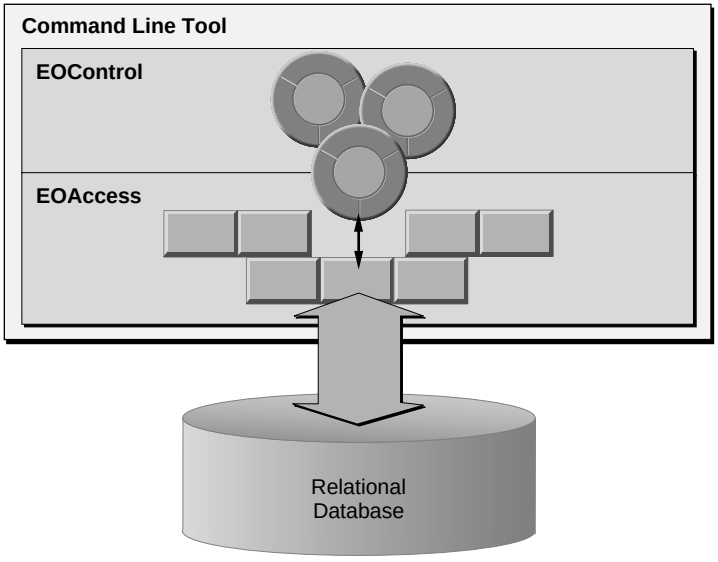


Figure 7. A Command-Line Program

Using a model, the *access layer* fetches rows of data from a database, creates enterprise objects from the fetched data, and registers the enterprise objects with the control layer. The *control layer* manages the graph of enterprise objects in memory, tracking changes to them and directing the access layer to commit those changes to the database when the program is ready to save.

An Application Kit Client/Server Application

The second type of application is a traditional client/server application in which desktop clients access a database running on a server. To the simple architecture of a command-line program, a desktop client adds a graphical user interface and two additional frameworks to support that interface (Figure 8).

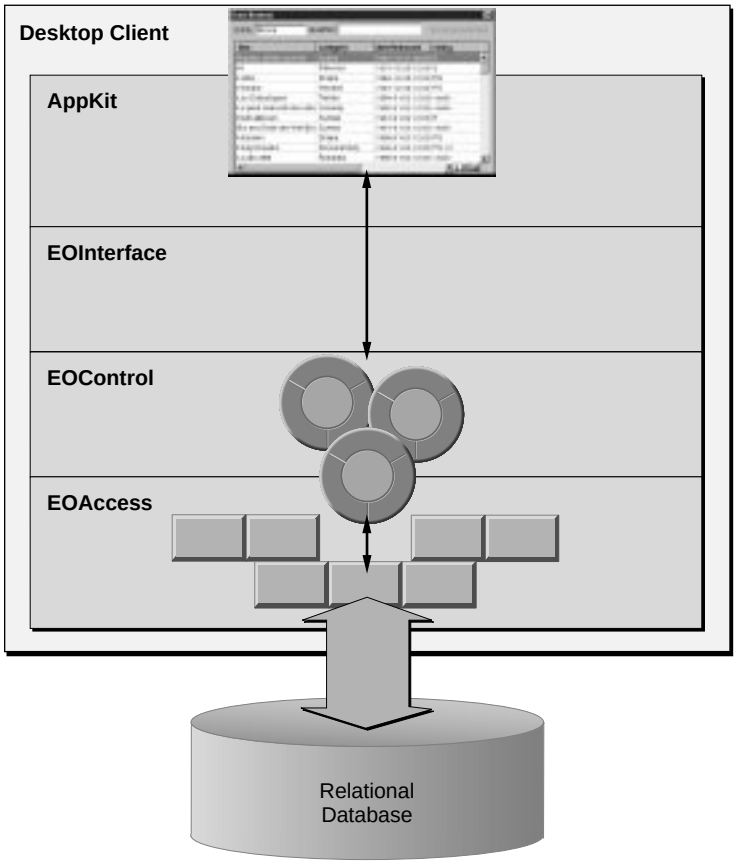


Figure 8. An Application Kit Client/Server Application

The *Application Kit* is a framework that provides the structure and user-interface controls (buttons, text fields, and tables, for example) for applications with graphical user interfaces. The Application Kit isn't a part of Enterprise Objects Framework; rather it's a fundamental component of the Yellow Box development environment. For more information on the Application Kit, see the "Introduction to the Application Kit" in the *Application Kit Reference*.

On the other hand, *interface layer* is a part of Enterprise Objects Framework. It maps data between the application's user interface and the control layer's graph of enterprise objects.

An HTML WebObjects Application

The third type of application is an HTML web application. Like the Application Kit client/server application, the HTML web application uses the access and control layers to fetch and manage enterprise objects. However, the HTML web application replaces the Application Kit-based user interface with a standard, HTML web page (Figure 9).

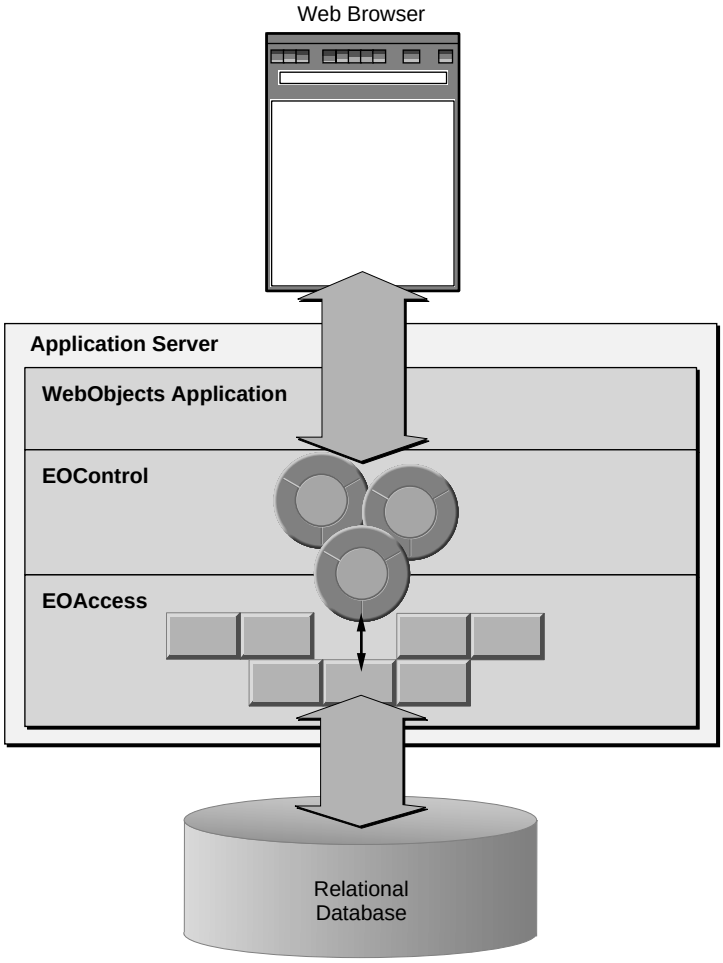


Figure 9. An HTML WebObjects Application

The *WebObjects* framework manages the user interface in an HTML web application. It combines the functionality of the Application Kit and EOInterface to provide an HTML-based presentation layer. Like the Application Kit, the WebObjects framework provides structure and user-interface elements. It also provides its own mechanism for transporting data between the control layer's graph of enterprise objects and a web page. WebObjects framework, like the Application Kit, is not a part of Enterprise Objects Framework. Rather, it's the fundamental component of the WebObjects product. For more information on the WebObjects framework, see the *WebObjects Developer's Guide*.

The Application Kit client/server application and the HTML web application are both client/server applications; but in the former, the client is fat, and in the latter, the client is thin. Put another way, the Application Kit application puts all (or most) of the application and business logic in the client, whereas the HTML web application puts all (or most) of the application and business logic in the application server.

In contrast, a web application with Java client distributes the logic more evenly across the client and the application server (Figure 10). This last type of application introduces the final Enterprise Objects Framework layer: the *distribution layer* keeps copies of enterprise objects in the application server synchronized with those in the client.

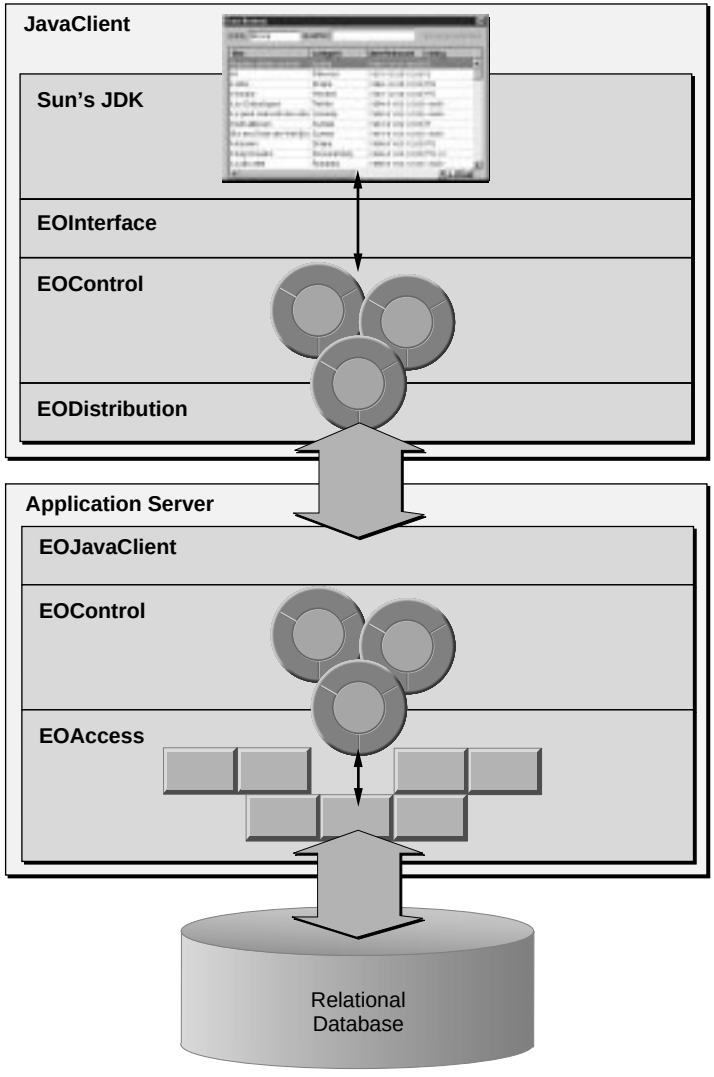


Figure 10. A Web Application with a Java Client

The kinds of applications you can build with Enterprise Objects Framework aren't limited to these four types; the types in this chapter are simply the most basic. They can be mixed and matched to create numerous other configurations. For example, you could build a

command-line application to act as a server for an Application Kit-based desktop client, distributing your business logic across the client and server the way you can with a web application and a Java client.

Enterprise Objects

So far, you've seen the components of an application that Enterprise Objects Framework provides. The component of an application that the Framework doesn't provide—the part that you write—is your application's business logic. Typically you code the bulk of this business logic in enterprise object classes.

An enterprise object is like any other object in that it couples data with the methods for operating on that data. However, an enterprise object class has certain characteristics that distinguish it from other classes:

- It has properties that map to *stored* or persistent data; an enterprise object instance typically corresponds to a single row or record in a database.
- It knows how to interact with other parts of the Framework to give and receive values for its properties.

Although you write the business logic, the Framework specifies how it gets invoked. In addition to providing classes that manage a graph of enterprise objects in memory, the control layer defines an API to which enterprise objects must conform. So you can concentrate on the parts of your enterprise object classes that are specific to your application, it also provides default implementations of most of this API.

To find out more about writing enterprise object classes, see the chapter "Designing Enterprise Objects" on page 65.

Chapter 2

Enterprise Objects Framework Viewed Through Its Classes

As you learned in the previous chapter, Enterprise Objects Framework is divided into layers. This chapter takes a closer look at the responsibilities of each layer. It does so by describing the classes in each layer and demonstrating the roles they play in the four most basic types of Enterprise Objects Framework applications:

- Command-line programs without a graphical user interface
- Application Kit client/server desktop applications where all the logic is in a client
- HTML WebObjects applications where all the logic is in the application server
- WebObjects applications with interactive Java clients where the logic is distributed between an application server and its clients

Classes in a Command-Line Program

A command-line program—one that doesn't have a graphical user interface—uses the most fundamental Enterprise Objects Framework classes (Figure 11). The same classes are used in all the most typical types of applications, but other types add other classes for interacting with the user interface.

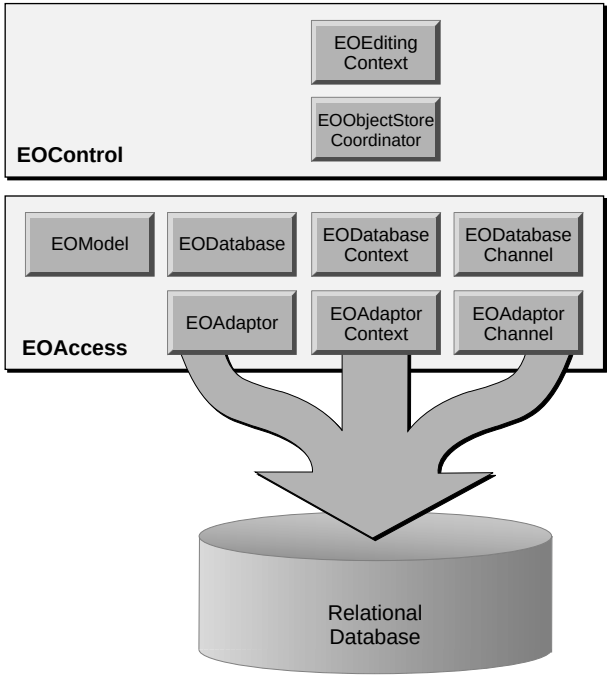


Figure 11. Framework classes in a Command-Line Program

The next sections introduce these classes, following the flow of data from the database, through the access layer, to the control layer.

The Access Layer

The access layer is the part of the Framework that interacts with the database. Its role in a command-line program is much the same as it is in all other types: it fetches rows of data from a database, creates enterprise objects from the fetched data, and registers the enterprise objects with the control layer—the next layer up in an application's architecture. Later, when the control layer has changes to save, it directs the access layer to write those changes to the database.

The access layer is divided into three functional groupings:

- The ***adaptor level*** that interacts with a database in terms of server-specific client libraries, providing server-independent database access to the rest of the Framework.
- The ***database level*** that creates full-fledged enterprise objects from database rows.
- *Modeling classes* that furnish database login information and a database-to-objects mapping.

Figure 12 shows how each grouping fits into the architecture of an application. The bottom row of classes (EOAdaptor, EOAdaptorContext, and EOAdaptorChannel) constitutes the adaptor level. The top row of classes in (EODatabase, EODatabaseContext, and EODatabaseChannel) constitutes the database level.

EOModel is one of the modeling classes. An EOModel object represents the whole database-to-objects mapping in entity-relationship terms, while other modeling classes correspond to components of that mapping. The adaptor level, database level, and modeling classes are described in greater detail in the following sections.

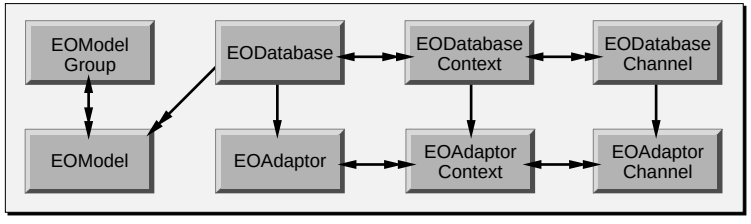


Figure 12. The Access Layer

The Adaptor Level

The adaptor level defines a server-independent interface for working with relational database systems. Figure 13 shows the adaptor level classes and the behaviors associated with each class.

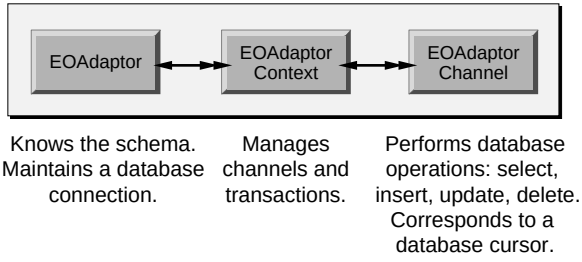


Figure 13. Adaptor Level

Server-specific subclasses encapsulate the behavior of database servers, thereby offering a uniform way of interacting with servers while still allowing applications to exploit their unique features. For example, the Framework provides the classes OracleAdaptor, OracleAdaptorContext, and OracleAdaptorChannel which implement the functionality specified in the adaptor level in terms of the Oracle client libraries. Together, the server-specific subclasses are referred to as an *adaptor*. For instance, the Oracle subclasses are collectively referred to as the Oracle adaptor.

The adaptor level deals with database rows packaged as NSDictionary objects. When an adaptor fetches from a relational database, it receives the raw data in whatever form the database client libraries provide. The adaptor then packages the data in dictionaries—one per database record. Each dictionary contains key-value pairs; the keys typically represents the name of a column, and the key's value corresponds to the data for the column in that particular row. Going the other way, the adaptor unpacks dictionaries into raw data that the server can accept whenever it needs to insert or update database rows.

The Database Level

The database level creates enterprise objects from the dictionaries retrieved by the adaptor level. It's also where *snapshotting* is performed. Snapshotting is used by Enterprise Objects Framework to manage updates. For caching, For updating, when an object is fetched from the database, a snapshot is taken of its state. A snapshot—an NSDictionary object—is consulted when you perform an update to verify that the data in the row to be updated has not changed since you fetched the object.

Figure 14 shows the database level classes and the behaviors associated with each class.

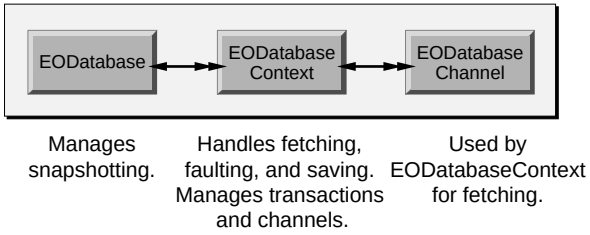


Figure 14. Database Level

Modeling Classes

The correspondence between an enterprise object class and stored data is established and maintained in a ***model***. A model defines, in entity-relationship terms, the mapping between enterprise object classes and a database. Figure 15 shows the modeling classes.

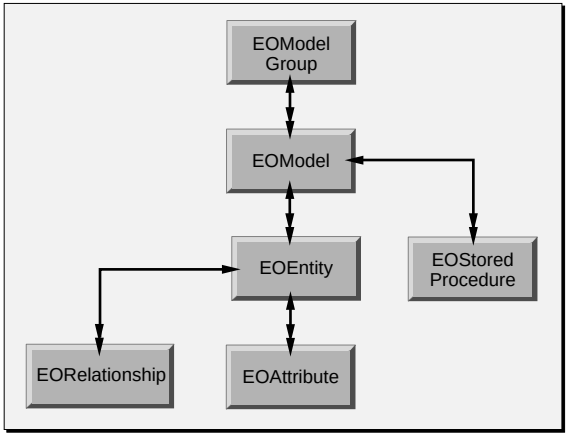


Figure 15. Modeling Classes

Most of the modeling classes represent components of the database-to-object mapping. The following table summarize the role of those classes:

Database Element	Model Object	Object Mapping
Data Dictionary	EOModel	—
Table	EOEntity	Enterprise object class
Row	—	Enterprise object instance
Column	EOAttribute	Enterprise object instance variable (class property)
Referential Constraint	EORelationship	Reference to another object

In addition to storing a mapping between the database schema and enterprise objects, an EOModel object stores information needed to connect to the database server. This connection information includes the name of the adaptor corresponding to your application's database server. An EOModel can also store information about a database's stored procedures (in EOStoredProcedure objects).

While a model can be generated programmatically at run time, the most common approach is to use the EOModeler application to create models and store them in files. The Framework knows how to initialize modeling objects from a model file. Simply by adding a model file to a project, you make the model's database-to-object mapping available to the Framework objects that need to reference it. All of the models available to an application are managed by an EOModelGroup object; see the EOModelGroup class specification in the *Enterprise Objects Framework Reference* for more information.

For a discussion of entity-relationship modeling and how it relates to Enterprise Objects Framework, see the Appendix "Entity-Relationship Modeling" on page 257.

The Control Layer

In a command-line program, the control layer's responsibility is to manage a graph of enterprise objects, tracking changes to them and directing the access layer to commit those changes to the database when the program is ready to save. The control layer classes that perform these duties are EOObjectStoreCoordinator and EOEditingContext. An EOObjectStoreCoordinator object manages interactions with the access layer, while EOEditingContext objects manage graphs of enterprise objects and track changes to those objects.

Interacting with the Access Layer

The control layer provides an infrastructure for managing enterprise objects that is independent of the storage mechanism being used. Put another way, you can use the control layer to interact with any external store: a relational database, a live data feed, or the file system, for example. To achieve this independence, the control layer defines an abstract class, EOObjectStore, whose subclasses represent "intelligent" sources and sinks of objects for EOEditingContexts. An object store is responsible for constructing and registering objects and for committing changes made in an editing context.

To allow applications to interact with more than one external store, the control layer provides two subclasses of EOObjectStore:

- *EOCooperatingObjectStore*, an abstract class that defines the basic API for object stores that work together to manage data from several distinct data repositories.

- *EOObjectStoreCoordinator*, a concrete class whose instances manages the interactions between editing contexts and cooperating object stores.

The access layer's EODatabaseContext is a concrete subclass of EOCooperatingObjectStore. Although different subclasses of EOObjectStore or EOCooperatingObjectStore can be defined for different types of external stores, most applications use an EOObjectStoreCoordinator and one or more EODatabaseContexts to access relational databases.

Object Graph Management and Change Tracking

An *object graph* is a group of related enterprise objects that represents an internally consistent view of an external store—typically a database. In a running application, the object graph is the central repository for data and business logic. An EOEditingContext object, which represents a single "object space" or "document" in an application, manages this in-memory graph of enterprise objects. All objects fetched from an external store are registered in an editing context with a global identifier (an EOGlobalID object) to uniquely identify each object. The editing context is then responsible for watching for changes in its objects and recording snapshots of them for object-based undo. For more information on change tracking, see the EOEditingContext class specification and the EOObserving interface (or protocol) specification in the *Enterprise Objects Framework Reference*.

Classes in an Application Kit Client/Server Application

An Application Kit client/server application uses an architecture similar to that in a command-line program, but it incorporates classes to synchronize values between enterprise objects and the application's user interface. This section describes the roles of those classes and how they fit into the application's architecture.

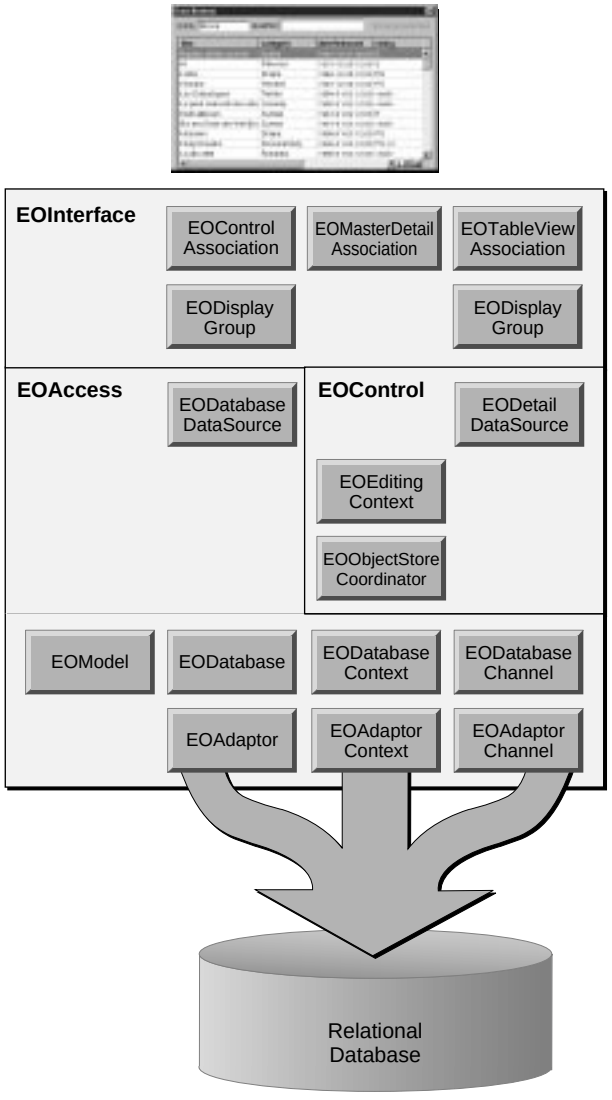


Figure 16. Classes in an Application Kit Application

User Interface Objects

In a traditional client/server application such as the one described in this chapter, Application Kit user interface objects (such as NSPopUpButtons, NSForms, NSTextFields, and NSTableViews) are used to display the values of enterprise objects. When values are edited

in the user interface, these same Application Kit objects are used to communicate the changes back to the enterprise objects. In Figure 16, Application Kit classes are represented by a screen capture of a real application's user interface.

The Interface Layer

The interface layer in an Application Kit application synchronizes data between the application's user interface (Application Kit objects) and the control layer's graph of enterprise objects. The relationship between user interface objects and enterprise objects is managed by EODisplayGroup objects. More precisely, display groups are used by EOAssociation objects to mediate between enterprise objects and the user interface. EOAssociations link a single user interface object to one or more class properties (keys) of the objects managed by a display group. The properties' values are displayed in the association's user interface object.

In the Interface layer, EOAssociation objects "observe" EODisplayGroups to make sure that the data displayed in the user interface remains consistent with enterprise object data. Display Groups interact with data sources, which supply them with enterprise objects.

Access and Control Layers

The roles of the access and control layers in an Application Kit application are the same as they are in a command-line application. However, in an Application Kit application, each layer supplies a *data source* for interacting with the interface layer.

A data source is a subclass of the EODataSource abstract class that presents an EODisplayGroup object with a standard interface to a store of enterprise objects. From the perspective of the EODisplayGroup to which a data source supplies enterprise objects, the actual mechanism used for storing data is of no concern; everything below the data source is effectively a "black box." The interface layer interacts with all data sources in the same way. A data source takes care of communicating with the external data store to fetch, insert, update, and delete objects.

For most database applications, data sources are instances of EODatabaseDataSource or EODetailDataSource (the data source classes supplied with the Framework). EODatabaseDataSource, defined in

EOAccess, provides an interface to the Framework's access layer and ultimately, to a relational database. However, the data source can be any object that is a subclass of the abstract class EODataSource. Thus, the user interface layer can be used independently from the access layer for other types of data sources, such as an array of objects constructed by an application, or objects fetched from a flat-file database or a newsfeed.

Data sources can be arranged in master-detail configurations to support master-detail displays. For example, suppose an application displays movie studios in one table and the movies for the selected studio in another table. Selecting a new studio updates the movies table to display the movies for the newly selected studio. To support this user interface, the application has a master data source for Studio objects and a detail data source for Movie objects. Based on a relationship between Studio and Movie (a studio has many movies), the detail data source limits its Movie objects to those associated with the Studio that's selected in the application's user interface.

Most often the master data source is an EODatabaseDataSource, while the detail is an EODetailDataSource. EODetailDataSource is provided by the control layer, and is a general purpose data source for master-detail configurations.

Classes in an HTML WebObjects Application

The major difference between an Application Kit application and an HTML web application is that the web application uses the WebObjects framework instead of the Application Kit. Figure 17 shows how the WebObjects framework provides a web application's presentation layer.

The WebObjects box is jagged at the top because not all of the WebObjects classes that participate in user interface management are illustrated. For example, the WebObjects framework provides user interface elements such as WOTextField and WOBrowser for generating web pages that users see in their browsers. None of these classes are shown. Rather, Figure 17 shows only the WebObjects framework class that acts as the go-between for Enterprise Objects Framework's control layer and WebObjects user interface: WODisplayGroup.

WODisplayGroup is analogous to the interface layer's EODisplayGroup. They have virtually the same APIs, but EODisplayGroup works with the interface layer's EOAssociation's and WODisplayGroup works with WebObjects framework elements.

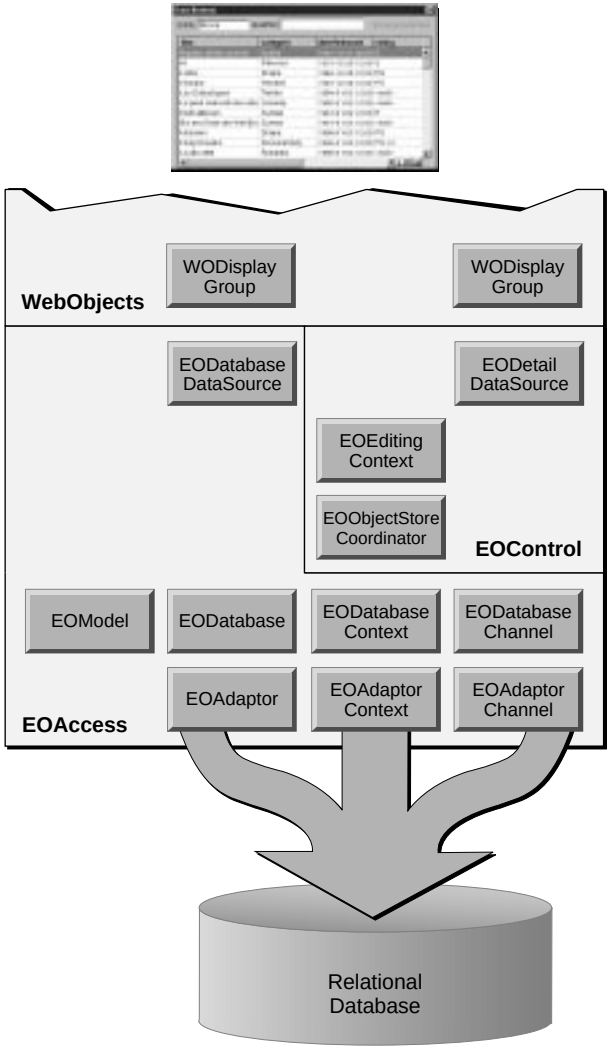


Figure 17. Classes in an HTML WebObjects Application

Note that you don't have to use WODisplayGroup's in your application. It's primarily useful for managing *batches* of enterprise objects, so users can page through the first set of objects, then the second, and so on. Additionally, WebObjects Builder has a lot of built-in support for WODisplayGroups. Using them, it's much simpler to construct the user interface for your web application, writing less code than you might if you were to use your own solution.

Classes in a Web Application with a Java Client

The main difference between a Java web application and the other types of applications is that a Java web application is distributed across an application server and Java clients. However, within this distributed structure, a Java web application combines architectural features from other application types into one. Figure 18 shows the classes in a web application. Notice that the access and control layers in the server side of the application are the same as they are in a command-line application. Similarly, the control and interface layers in the client side of the application are the same as they are in an Application Kit application.

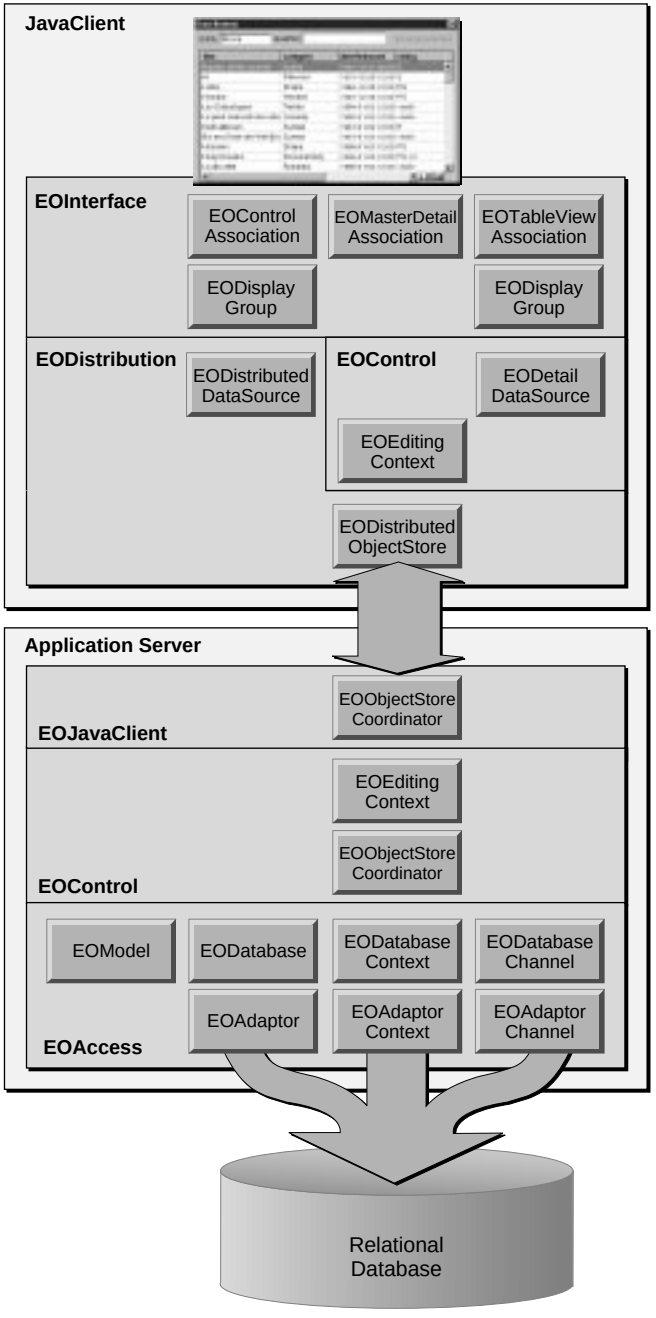


Figure 18. Classes in a Web Application with a Java Client

To support distributing the application across the server and clients, the distribution layer is inserted. There's a server side and a client side to the distribution layer. The server side, provided by the EOJavaClient framework, is written in Objective-C. The client side, provided by the com.apple.client.eodistribution package, is written in Java. Together, the two sides of the distribution layer handle the communication between the application server and the user's Java client.

The Distribution Layer

The distribution layer provides *channels* through which the application server and Java clients communicate. The default channel is an HTTP channel, but you can write your own channel to use a different communication protocol (CORBA, for example). Use of the channel is completely transparent; the EODistributionContext on the server side and the EODistributedObjectStore on the client side handle all the interaction with the channel. So unless you are writing your own channel, you don't need to learn about EODistribution's channel and related classes.

The EODistributionContext class encodes data to send to the client and decodes data it receives from the client. Additionally, it keeps track of state necessary to keep the client and server in sync.

EODistributedObjectStore is a concrete subclass of the control layer's abstract EOObjectStore class. It merely incorporates knowledge of the distribution layer's channel so it can forward messages it receives from the server to its editing context as well as messages going the other way.

EODistributedDataSource is a concrete subclass of the control layer's abstract EODataSource class. Whereas the access layer's EODatabaseDataSource fetches using an EODatabaseContext, an EODistributedDataSource fetches using an editing context (which in turn, forward the fetch request to the server where the request is ultimately serviced by an EODatabaseContext).

Client-Side APIs

The EOControl and EOInterface classes on the client side of a Java web application are actually not the same classes as the ones on the server side. There are two different versions of each framework.

On the server side, you can write your application in either Objective-C or Java. The Java APIs (com.apple.yellow.eoaccess, com.apple.yellow.eocontrol, and com.apple.yellow.eointerface) are actually *wrappers* for the corresponding Objective-C frameworks. When you invoke a Java method from one of the com.apple.yellow packages, the message is forwarded across Apple's Java bridge to a corresponding Objective-C object. On the client side, however, the APIs (com.apple.client.eocontrol, com.apple.client.eodistribution, and com.apple.client.eointerface) are implemented in pure Java.

Note: In the client, you don't have the option of writing Objective-C code.

Conceptually, the classes in com.apple.yellow and com.apple.client are the same and generally their APIs are identical. However, there are some differences, the most significant of which are that:

- EOEditingContext in the client doesn't support undo and redo.
- The EOAssociation subclasses in the client are implemented in terms of Swing (the presentation component of Sun's JDK) instead of the Application Kit.

Framework Dependencies

The architectural depictions of Enterprise Objects Framework in the previous sections present the ordering of the Framework components in terms of the conceptual data flow in the system. Another way to look at Enterprise Objects Framework is in terms of the structural dependencies of the components on one another.

Figure 19 shows the relationships between the Framework's Objective-C frameworks (and also for the corresponding Java versions).

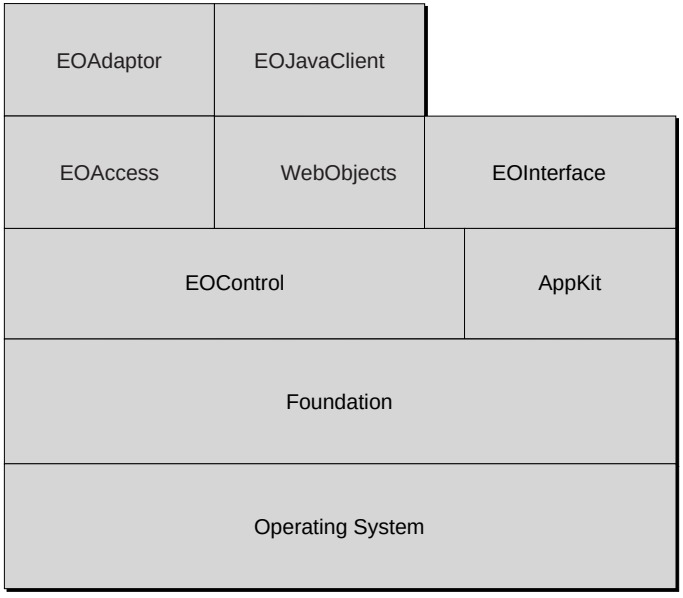


Figure 19. Objective-C Framework Dependencies

The control layer is the lowest layer in the Framework. It can be thought of as an extension of Foundation in that it defines generic core functionality, such as key-value access and object change notification. The control layer centers around EOEditingContext, a subclass of EOObjectStore that manages enterprise objects in memory.

The access layer extends the control layer by implementing an EOObjectStore for relational databases, EODatabaseContext. WebObjects framework too depends on EOControl, because it provides an EOEditingContext with each WOSession object.

The interface layer extends the control layer and the Application Kit by adding bindings between enterprise objects and the user interface. This keeps the values of enterprise objects in sync with their display in the user interface.

Each concrete adaptor (ODBCAdaptor and OracleAdaptor, for example) extends the access layer by implementing concrete subclasses of the access layer's adaptor level classes (EOAdaptor, EOAdaptorContext, and EOAdaptorChannel).

Finally, the server-side EOJavaClient framework extends WebObjects by providing a WOComponent for displaying Java interfaces.

Figure 20 shows the relationships between the Framework's pure Java packages (for writing Java client applications).

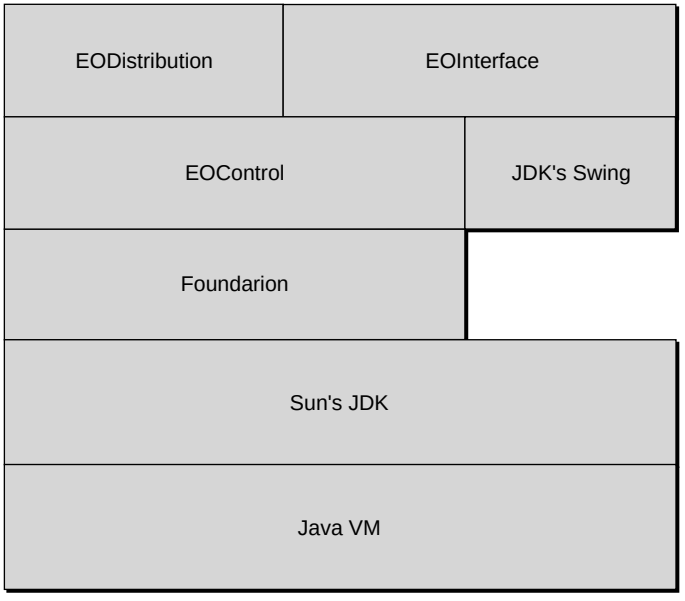


Figure 20. Java Package Dependencies

Again the control layer is the lowest layer in the Framework. The interface layer extends the control layer and Swing (the presentation layer of Sun's JDK). Finally, the distribution layer extends the control layer by implementing an EOObjectStore for communicating to an application server through a channel.

Part II

Enterprise Object Design

Chapter 3

Designing Enterprise Objects

The Enterprise Objects Framework and the applications you build with it revolve around the enterprise objects that you design. Designing these objects, then, is in many ways the essence of creating an Enterprise Objects Framework application. This chapter explains the mechanics of designing enterprise objects, describes their structure and interaction with the Framework, and explains how you can take advantage of features provided by the Framework.

Designing an enterprise object entails three major steps:

- Designing your schema
- Modeling the enterprise object
- Implementing the enterprise object

This chapter describes the activities that occur during each. The EOModeler application plays a part in all stages of enterprise object design, so this chapter refers frequently to *Enterprise Objects Framework Tools and Techniques*, where using EOModeler is documented.

This chapter uses selections from the Enterprise Objects Framework on-line examples to explain the principles of designing an enterprise object. In particular, the chapter focuses on the Customer entity in the Rentals database. A customer is a video store member who's authorized to rent videos.

Designing Your Schema

If you're working with an existing database, its schema will dictate many of the decisions you make in designing your enterprise objects. If you're designing your database at the same time as your enterprise objects, however, you can let each design influence the other as they're developed and before you implement them. Be sure to keep both designs in mind as you work; decisions you make about the database design can affect your enterprise object design, and vice versa. This chapter doesn't address issues of database design itself, but the information presented here can help you to create a design that will work effectively with the Enterprise Objects Framework.

Defining the Model

The work of writing enterprise objects typically begins in EOModeler. You make the following decisions for your enterprise object in EOModeler:

- Should your enterprise object be an EOGenericRecord or a custom class?
- What database attributes do you want to include as properties in your class?
- What data types should the class properties be?
- What relationships does your enterprise object have with other objects?
- What referential integrity rules should you specify for the relationships in your enterprise object?
- Does your enterprise object class have inheritance relationships with other enterprise object classes?

These issues are discussed in the following sections.

EOGenericRecord or Custom Class?

Enterprise Objects Framework provides a "default" enterprise object class, EOGenericRecord. An EOGenericRecord can take on values for any properties defined in your application's model (see the section "Accessing an Enterprise Object's Data" on page 82 for more discussion of this), but implements no custom behavior. EOGenericRecord objects can hold simple values as well as refer to other enterprise objects through relationships defined in the model.

The criterion for deciding whether to make your enterprise objects custom classes or to simply use the EOGenericRecord class is behavior. One of the main reasons to use the Enterprise Objects Framework is to associate behavior with your persistent data. Behavior is implemented as methods that "do something," as opposed to merely returning the value

for a property. Since the Framework itself handles most of the behavior related to persistent storage, you can focus on the behavior specific to your application.

Because the Customer class in the Rentals database has specialized behavior—for example, it rents units—it needs to be a custom class.

Which Attributes Should Be Class Properties?

By default, EOModeler marks all of the attributes read in from the database as class properties. When an attribute is marked as a class property, it means that the attribute will be included in your class definition when you generate source files for the class, and that it will be fetched and passed to your enterprise object when you create instances from the database.

The attributes you mark as class properties should only be ones whose values are meaningful in the object graph that's created when you fetch objects from the database. Attributes that are essentially database artifacts shouldn't be marked as class properties. For example:

- As a general rule, foreign and primary keys shouldn't be included in your enterprise object as class properties. The only exception to this rule is when the key has meaning to the user (such as a credit card number) and therefore must be displayed in the user interface.
- Relationships that are just used in EOModeler as a vehicle for flattening attributes from another entity shouldn't be included as class properties. For example, suppose an Employee class flattens address properties (**streetAddress**, **city**, and so on) from an Address entity and that Employee includes these flattened attributes as class properties. The relationship from Employee to Address doesn't need to be a class property if it's otherwise not needed. For more discussion of this topic see "How Should Your Enterprise Object Manage Relationships with Other Objects?" on page 72.
- Relationships that represent an entity's relationship to an intermediate join table (also known as a *correlation* table) shouldn't be included in your enterprise objects as class properties (unless they contain data that's meaningful in your application).

For example, in the Movie database, the Director table acts as an intermediate table between Movie and Talent and exists purely to define that relationship. It has no data besides its foreign keys. Because of this, you never need to fetch instances of Director into your application. However, it makes sense to specify a relationship between Movie and Director and between Director and Talent, and to flatten that second relationship to give Movie access to the Talent table. The flattened relationship, possibly named **directors**, can then be marked as a class property, because it contains objects that should be included in the object graph.

Although Director contains no data besides its foreign keys, some intermediate tables do. For example, the MovieRole table acts as an intermediate table between Movie and Talent, and it includes the attribute **roleName**. Because of this, it's likely that if your enterprise object had a relationship to MovieRole, you'd want to include that relationship as a class property to be able to access the value of **roleName**.

What Data Types Should Your Properties Be?

When you create a new model, Enterprise Objects Framework maps external database data types to the following standard value classes:

Java	Objective-C
String	NSString
BigDecimal (java.math)	NSDecimalNumber
Number	NSNumber
NSGregorianDate	NSCalendarDate
NSData	NSData

Additionally, you can map external data types to custom value classes defined by your application. When you're working with custom data, you'll typically want to convert binary data into a meaningful form. However, since you define its form, you have to convert it yourself. The Framework allows you to define a custom class whose instances are initialized with binary or string data; this prevents your accessor methods from having to explicitly convert the data, and allows other objects to

access your enterprise object's property in its intended form rather than as an NSData object. For more discussion of this subject, see the chapter "Advanced Enterprise Object Modeling" on page 105.

Working with Numeric Values

There are basically two different choices for representing numeric values in your model:

- Numbers that correspond to money values should be represented in your model as BigDecimals (or NSDecimalNumbers in Objective-C). This is because in the Enterprise Objects Framework, numeric database values are stored in one of two types of objects: Number and its subclasses or BigDecimal (NSNumber or NSDecimalNumber in Objective-C). When you use Number, values are limited to double precision, and operations are inexact. Using BigDecimal provides high precision and smooth conversions between strings, BigDecimals, and money.

- Numbers that represent integer or floating point values in your database should be declared as Number objects (NSNumbers in Objective-C) in your model. You then use the Attribute Inspector to specify the scalar type to which the Number will be coerced. For example, if you're working with scientific data, you should represent it as a Number that will be coerced to a **double**.

Conversion of Numeric Values

When the Framework passes a Number value (NSNumber in Objective-C) to your object, the value can be converted to the corresponding scalar (numeric) type if your accessor method or instance variable requires it (for more information, see "Accessing an Enterprise Object's Data" on page 82). For example, suppose your enterprise object defines these accessor methods:

```
public void setAge(int age)
public int age
```

For the **setAge** method, the Number value for the "age" key is converted to an **int** and passed as *age*. Similarly, the return value of the **age** method is converted to an Number.

How Should Your Enterprise Object Manage Relationships with Other Objects?

In EOModeler you can specify relationships between entities. For example, in the Rentals database in the on-line examples, the Member entity can have several relationships to other entities, including:

- To-one relationship to Customer
- To-one relationship to CreditCard
- To-many relationship to Guest
- To-many relationship to Rental
- To-many relationship to Fee

You can use relationships to access data in the destination entity from the source entity.

When you include relationships as class properties, to-one relationships are represented as references in the object graph and to-many relationships are represented as NSArrays. You can see this more clearly if you look at the way Customer's to-one relationship to CreditCard and to-many relationship to Rental are represented as instance variables in the Customer class:

```
// Reference to a CreditCard object in the object graph
protected CreditCard creditCard;

// Array of Rental objects
protected NSMutableArray rentals;
```

For the most part, you access the data in other objects by using relationship properties to traverse the in-memory object graph in your running application. For example, the following statement uses Customer's **creditCard** relationship to access the **authorizationDate** property in the CreditCard object:

```
date = customer.creditCard().authorizationDate();
```

Likewise, the following statement uses the Customer's **rentals** relationship to return an NSArray containing a customer's Rental objects:

```
rentalArray = customer.rentals();
```

Referential Integrity

When your enterprise object has relationships with other objects, you can use EOModeler to define rules to govern those relationships. Each of the rules possible are described briefly in the following sections. For more information on how to set these options, see the book *Enterprise Objects Framework Tools and Techniques*.

Optionality

Optionality refers to whether a relationship is optional or mandatory. For example, you can require all departments to have a location (mandatory), but not require that every employee have a manager (optional).

Delete Rules

This refers to the rules that should be applied to an entity that's involved in a relationship when the source object is deleted. For example, you might have a department with multiple employees. When a user tries to delete the department, you can:

- Delete the department and remove any back reference the employee has to the department (nullify).
- Delete the department and all of the employees it contains (cascade).
- Refuse the deletion if the department contains employees (deny).
- Delete the department but *do not* remove any back reference the employee has to the department. *Use this option with caution as dangling references may result.*

Owns Destination

You can set a source object as owning its destination objects. When a source object owns its destination objects and you remove a destination object from the source object's relationship array, you're also deleting it from the database (alternatively, you can transfer it to a new owner). This is because ownership implies that the owned object can't exist without an owner—for example, line items can't exist outside of a purchase order.

By contrast, you might have a department object that doesn't own its employee objects. If you remove an employee from a department's employees array, the employee continues to exist in the database, but its department variable is set to **null** (or **nil** in Objective-C). If you really intend to delete the employee from the database, you'd have to do it explicitly.

Propagates Primary Key

You can also specify in EOModeler that the primary key of the source entity should be propagated to newly inserted objects in the destination of the relationship. This is used for a to-one owning relationship, where the owned object has the same primary key as the source. For example, in the Movies database the TalentPhoto entity has the same primary key as the entity that owns it, Talent. In this scenario, if you create a new Talent object and the Talent doesn't already have a TalentPhoto, the Framework automatically creates a TalentPhoto object for you.

Mapping an Entity Across Multiple Tables

In certain special cases you may decide to use EOModeler to "flatten" attributes from one entity into another. In general you should only flatten attributes across one-to-one relationships (like Employee to Address) where the destination entity is never fetched directly. Otherwise, you run the risk that the values of flattened attributes can get out of sync with the most current view of data in your application.

Some examples of good uses of flattened attributes are as follows:

- Combining multiple tables to form a logical unit.

  For example, you might have employee data that's spread across multiple tables such as Address, Benefits, and so on. If you have no need to access these tables individually (that is, if you'd never create an Address object since the address data is always subsumed in Employee), then it makes sense to flatten attributes from those entities into Employee.

- Implementing vertical inheritance mapping.

  For example, the Member class in the RentalsInheritance model has two flattened attributes: **firstName** and **lastName**. It flattens these attributes from the Customer entity. Customer is a parent entity of Member and Guest that provides attributes common to both. Because Customer is an abstract entity and is therefore never instantiated in the object graph, the only way to access Customer's data is to flatten the appropriate attributes into its sub-entities. The relationship between Member, Guest, and Customer is an example of vertical inheritance mapping—for more discussion of this topic, see the chapter "Advanced Enterprise Object Modeling" on page 105.

- If your application (or the property in question) is read-only.

When you use flattened attributes, you don't need to include the relationship as a class property—there's no need to since the data it would be used to access is already included in the source entity.

For more discussion of this topic, see the book *Enterprise Objects Framework Tools and Techniques*.

What about Inheritance?

You can use the Advanced Entity Inspector in EOModeler to specify an entity's parent entity. For example, the Customer entity is a parent to both Member and Guest in the RentalsInheritance model. For more discussion of the different approaches you can use for inheritance, see the chapter "Advanced Enterprise Object Modeling" on page 105.

Implementing an Enterprise Object

As discussed in the section "EOGenericRecord or Custom Class?" on page 68, one of the first decisions you need to make about an enterprise object is whether you want it to be an EOGenericRecord or a custom class. Use EOGenericRecords to represent enterprise objects that don't require custom behavior, and create custom classes to represent enterprise objects that do.

Enterprise Objects Framework interacts with generic records and custom classes the same way. It defines the set of methods for supporting operations common to all enterprise objects and uses only those methods. The EOEnterpriseObject interface (or informal protocol in Objective-C) specifies the complete set. It includes methods for initializing instances, announcing changes, setting and retrieving property values, and performing validation of state.

You rarely need to implement the EOEnterpriseObject interface from scratch. The Framework provides default implementations of all its methods. In Java, the class EOCustomObject implements the interface, so your custom classes should inherit from it. In Objective-C, categories on the root class, NSObject, provide default implementations. Regardless of what language you choose, some of the default method implementations are for custom classes to override, but most are meant to be used as defined by the Framework. Many methods are used internally by the Framework and are rarely invoked by your application code. For more information on EOEnterpriseObject and its methods, see the interface specification in *Enterprise Objects Framework Reference.*

The remainder of this chapter describes implementing custom classes.

Generating Source Files

The easiest way to create a custom enterprise object is with EOModeler's Generate Java Files, Generate ObjC Files, and Generate Client Java Files commands. These commands take the attributes and relationships you've defined in your model, and use them to generate source code for a corresponding enterprise object class.

You can generate the source code in Java (Generate Java Files) or Objective-C (Generate ObjC Files). The Generate Client Java Files command creates source code for classes that will be used in the client side of a distributed WebObjects application.

If you choose Java as your language, EOModeler creates a **.java** file for your class. If you choose Objective-C, it creates both a header (**.h**) and an implementation (**.m**) file. No matter what the language, the source files created by EOModeler are completely functional.

Superclass

EOModeler assumes that your custom classes descend from EOCustomObject (or NSObject in Objective-C) so that they inherit the Framework's default implementations of the EOEnterpriseObject interface.

Instance Variables

EOModeler's generated source code files contain instance variables for all of the corresponding entities class properties, both attributes and relationships alike. For example, the following code excerpts show the instance variables created for the Member class:

In Java, (from **Member.java**)

```
protected String city;
protected NSGregorianDate memberSince;
protected String phone;
protected String state;
protected String streetAddress;
protected String zip;
protected Number customerID;
protected String firstName;
protected String lastName;
protected CreditCard creditCard;
protected NSMutableArray rentals;
```

In Objective-C (from **Member.h**)

```
NSString *city;
NSCalendarDate *memberSince;
NSString *phone;
NSString *state;
NSString *streetAddress;
NSString *zip;
NSNumber *customerID;
NSString *firstName;
NSString *lastName;
CreditCard *creditCard;
NSMutableArray *rentals;
```

All of the instance variables correspond to attributes in the Member entity except for **creditCard** and **rentals**, which correspond to relationships.

Primary key generation

Enterprise objects don't have to declare instance variables for primary key and foreign key values. The Framework manages primary and foreign keys automatically.

The default mechanism for assigning unique primary keys is provided with the EOAdaptorChannel method **primaryKeyForNewRowWithEntity** (or **primaryKeyForNewRowWithEntity:** in Objective-C). If you need to provide a custom mechanism for assigning primary keys, you can implement the EODatabaseContext delegate method **databaseContextNewPrimaryKey** (or **databaseContext:newPrimaryKeyForObject:entity:** in Objective-C). Using either of these two techniques means you don't need to store the primary key in your enterprise object. For more discussion of primary keys, see the chapter "Answers to Common Design Questions" on page 221.

Note that Enterprise Objects Framework doesn't support modifiable primary key values—you shouldn't design your application so that users can change a primary key's value. If you really need this behavior, you have to implement it by deleting an affected object and reinserting it with a new primary key.

Writing Accessor Methods

When you generate source files for custom classes in EOModeler, the resulting files include default accessor methods. Accessor methods let you set and return the values of your class properties (instance variables). For example, here are some of Customer's accessor methods:

In Java, (from **Member.java**)

```
public NSGregorianDate memberSince() {
  willChange();
  return memberSince;
}

public void setMemberSince(NSGregorianDate value) {
  willChange();
  memberSince = value;
}

public CreditCard creditCard() {
  willChange();
  return creditCard;
}

public void setCreditCard(CreditCard value) {
  willChange();
  creditCard = value;
}

public NSArray rentals() {
  willRead();
  return rentals;
}

public void setRentals(NSMutableArray value) {
  willChange();
  rentals = value;
}

public void addToRentals(Rental object) {
  willChange();
  rentals.addObject(object);
}

public void removeFromRentals(Rental object) {
  willChange();
  rentals.removeObject(object);
}
```

In Objective-C (from **Member.m**)

```
- (void)setMemberSince:(NSCalendarDate *)value
{
  [self willChange];
  [memberSince autorelease];
  memberSince = [value retain];
}
- (NSCalendarDate *)memberSince { return memberSince; }

- (void)setCreditCard:(CreditCard *)value
{
  // a to-one relationship
  [self willChange];
  [creditCard autorelease];
  creditCard = [value retain];
}
- (CreditCard *)creditCard { return creditCard; }

- (void)addToRentals:(Rental *)object
{
  // a to-many relationship
  [self willChange];
  [rentals addObject:object];
}
- (void)removeFromRentals:(Rental *)object
{
  // a to-many relationship
  [self willChange];
  [rentals removeObject:object];
}
- (NSArray *)rentals { return rentals; }
```

Features introduced by these code excerpts are discussed in the following sections.

Note: You don't have to provide accessor methods for your object's properties. The Framework can access your object's properties directly (through its instance variables). For more information, see the section "Accessing an Enterprise Object's Data" on page 82."

Change Notification

In the above code excerpts from Member source files, you can see that each of the "set" methods includes an invocation of the **willChange** method.

In Enterprise Objects Framework, objects that need to know about changes to an enterprise object register as observers for change notifications. When an enterprise object is about to change, it has the responsibility of posting a change notification so that registered observers are notified. To do this, enterprise objects should invoke the method **willChange** prior to altering their state. This is invoked by default in generated source code's "set" methods, but whenever you add your own methods that change the object's state, you need to remember to include an invocation.

An implementation of **willChange** is provided in EOCustomObject (or NSObject in Objective-C), so you don't have to implement it yourself. For more information on change notification, see the specification for the control layer's EOObserverCenter class in *Enterprise Objects Framework Reference*.

Note: If you're using EOGenericRecord, you don't have to worry about invoking **willChange**. As the default enterprise object class, EOGenericRecord handles change notifications as part of its normal behavior.

Faulting

Similar to the way "set" methods invoke the **willChange** method, Java "get" methods include an invocation of **willRead**. This method is part of Enterprise Objects Framework's *faulting* mechanism for postponing an object's initialization until its actually needed. Faulting is also provided with Objective-C, but its implementation is different and doesn't require the **willRead** method.

In Java, the **willRead** method and a handful of others are defined in the Framework's EOFaulting interface. EOFaulting is a part of EOEnterpriseObject, so your custom classes inherit a default implementation of it. The default implementation of **willRead** checks to see if the receiver has already been fully initialized. If it hasn't been, it fills the object with values fetched from the database. Before your application attempts to message an object, you must ensure that it has its data. To do this, enterprise objects invoke the method **willRead** prior to any attempt to access the object's state, most typically in "get" methods. Enterprise Objects don't have to invoke **willRead** in "set" methods, because the **willChange** method invokes **willRead** for you.

For more information on faulting, see the interface specification for EOFaulting in *Enterprise Objects Framework Reference*. For more information on Objective-C's faulting mechanism, see the EOFault class specification.

Accessing Data through Relationships

Relationships and flattened properties are treated no differently than properties based on the entity's original attributes.

For example, Customer can use its **creditCard** relationship to traverse the object graph and change values in the CreditCard object. If you want to access information about a Customer's credit card, your code can do something like this:

In Java:

```
customer.creditCard().limit();
```

In Objective-C:

```
[[member creditCard] limit];
```

You can also modify attributes in the object graph regardless of what table they came from. For example, suppose that Customer's **streetAddress** attribute was flattened from an Address entity. Customer could have a **setStreetAddress** method that modified the attribute, even though it's not actually stored in the Customer table.

Accessing an Enterprise Object's Data

In implementing your enterprise object classes, you want to focus on the code that's unique to your application, not on code that deals with fitting your objects into the Framework. To this end, the Framework uses a standard interface for accessing an enterprise object's properties and provides a default implementation that takes advantage of methods you're likely to write for your own use anyway.

This data transport mechanism is defined by the EOKeyValueCoding interface (or informal protocol in Objective-C). It specifies that properties of an object are accessed indirectly by name (or *key*), rather than directly through invocation of an accessor method or as instance variables.

EOKeyValueCoding is incorporated in the EOEnterpriseObject interfaces, so your custom classes automatically inherit key-value coding behavior that is sufficient for most purposes.

The basic methods for accessing an object's values are **takeValueForKey** and **valueForKey** (**takeValue:forKey:** and **valueForKey:** in Objective-C), which set or return the value for the specified key, respectively. The default implementations use the accessor methods normally implemented by objects (or to access instance variables directly if need be), so that you don't have to write special code simply to integrate your objects into the Enterprise Objects Framework.

There are corresponding methods **takeStoredValueForKey** and **storedValueForKey** (**takeStoredValue:forKey:** and **storedValueForKey:** in Objective-C) are similar, but they're considered to be private to the Framework and to the enterprise object class that provides them. The stored value methods are for use by the Framework for transporting data to and from trusted sources. For example, **takeStoredValueForKey** is used to initialize an object's properties with values fetched from the database, whereas **takeValueForKey** is used to modify an object's properties to values provided by a user.

Public Access with the Basic Methods

When accessing an object's class properties, the default implementations of the basic key-value coding methods use the class definition. They look for "set" methods in the following sequence:

```
set<i>Property</i>
_set<i>Property</i>
```

where *property* is the name of the property as in **lastName**. Note that the first letter of the property name is made uppercase as in **setLastName**.

Similarly, the key-value coding methods look for "get" methods in the following sequence:

```
get<i>Property</i>
property
_get<i>Property</i>
<i>_property</i>
```

If no accessor methods can be found, the key-value coding methods look for instance variables containing the property's name and sets or retrieves their value directly. The search order for instance variables is as follows:

```
property
_property
```

where *property* is the name of the property as in **lastName**.

By using the accessor methods you normally define for your objects, the default implementations of the key-value coding methods allow you to concentrate on implementing custom behavior. They also allow your objects to determine how their properties are accessed. For example, your Employee class can define a **salary** method that just returns an employee's salary directly or calculates it from another value.

Note: You shouldn't use "set" methods to perform validation. Rather, you should use the validation methods, described in "Performing Validation" on page 87.

Private Access with the Stored Value Methods

The stored value methods, **takeStoredValueForKey** and **storedValueForKey**, are used by the framework to store and restore an enterprise object's properties, either from the database or from an in-memory snapshot (for undoing changes to an object and for synchronizing objects between child and parent editing contexts in a nested configuration, for example).This access is considered private to the enterprise object and is invoked by the Framework to effect persistence on the object's behalf. On the other hand, the basic key-value coding methods are the public API to an enterprise object. They are invoked by clients external to the object, such as for interactions with the user interface or with other enterprise objects.

Like the basic key-value coding methods, the stored value methods access an object's properties by invoking property-specific accessor methods or by directly accessing instance variables. However, the stored value methods use a different search order for resolving the property key: they search for a private accessor first (a method beginning with an underbar), then for an instance variable, and finally for a public accessor.

Enterprise object classes can take advantage of this distinction to simply set or get values when properties are accessed through the private API (on behalf of a trusted source) and to perform additional processing when properties are accessed through the public API. Put another way, the stored value methods allow you bypass the logic in your public accessor methods, whereas the basic key-value coding methods execute that logic.

The stored value methods are especially useful in cases where property values are interdependent. For example, suppose you need to update a total whenever an object's **bonus** property is set:

In Java:

```
void setBonus(double newBonus) {
  willChange();
  _total += (newBonus - _bonus);
  _bonus = newBonus;
}
```

In Objective-C:

```
- (void)setBonus:(double)newBonus {
  [self willChange];
  _total += (newBonus - _bonus);
  _bonus = newBonus;
}
```

This total-updating code should be activated when the object is updated with values provided by a user (through the user interface), but not when the **bonus** property is restored from the database. Since the Framework restores the property using **takeStoredValueForKey** and since this method accesses the **_bonus** instance variable in preference to calling the public accessor, the unnecessary (and possibly expensive and harmful) recomputation of **_total** is avoided. If the object actually wants to intervene when a property is set from the database, it has two options:

- Implement **_setBonus** (**_setBonus:** in Objective-C).
- Replace the Framework's default stored value search order with the same search order used by the basic methods. To do this, in your custom class, implement the static method **useStoredAccessor** to return **false**. (In Objective-C, override the class method **useStoredAccessor** to return NO.)

Error Handling methods

EOKeyValueCoding defines two additional methods for handling error conditions: **handleQueryWithUnboundKey** and **handleTakeValueForUnboundKey** (**handleQueryWithUnboundKey:** and **handleTakeValue:forUnboundKey:** in Objective-C). The default implementation of the key-value coding methods invoke these methods when they receive a key for which they can find no accessor methods or instance variables. The default implementations throw exceptions, but you can override them to handle the error in a way that's appropriate for your custom class.

Writing Derived Methods

The source files generated by EOModeler provide a basic implementation that doesn't go beyond the functionality provided by an EOGenericRecord. But you can use the files as a basis for adding behavior to your enterprise object.

One kind of behavior you might want to add to your enterprise object class is the ability to apply a filter to an object's to-many relationship property and return a subset of the destination objects. For example, you could have an **overdueRentals** method in the Customer class that returns the subset of a customer's rentals that are overdue:

In Java:

```
public NSArray overdueRentals() {
  EOQualifier qualifier;

  qualifier = new EOKeyValueQualifier(
      “isOverdue”,
      EOQualifier.QualifierOperatorEqual,
      new Boolean(true));

  return EOQualifier.filteredArrayWithQualifier(
      allRentals(),
      qualifier);
}
```

In Objective-C:

```
- (NSArray *)overdueRentals {
  EOQualifier *qualifier;

  qualifier = [[[EOKeyValueQualifier alloc]
      initWithKey:@"isOverdue"
      operatorSelector:EOQualifierOperatorEqual
      value:[NSNumber numberWithBool:YES]autorelease];

  return [[self allRentals]
      filteredArrayUsingQualifier:qualifier];
}
```

The **overdueRentals** method creates a qualifier to find all of a customer's overdue rentals. The qualifier is performed in memory against the complete set of a customer's rentals.

With the addition of this method, Customer has a "get" accessor method with no corresponding "set" method. A **setOverdueRentals** method doesn't make sense because **overdueRentals** is derived. However, since Enterprise Objects Framework invokes accessor methods automatically, you may wonder if the absence of a **setOverdueRentals** method can cause problems. The answer is no, you don't have to define a corresponding set method. Enterprise Objects Framework will not attempt to invoke **setOverdueRentals** unless you bind the **overdueRentals** property to an editable user interface object.

Performing Validation

A good part of your application's business logic is usually validation—for example, verifying that customers don't exceed their credit limits, that return dates don't come before their corresponding check out dates, and so on. In your enterprise object classes, you implement methods that check for invalid data, and the framework automatically invokes them before saving anything to the database.

The EOValidation interface (informal protocol in Objective-C) defines the way that enterprise objects validate their values. The validation methods can check for illegal value types, values outside of established limits, illegal relationships, and so on. Like EOKeyValueCoding, EOValidation is incorporated in the EOEnterpriseObject interface, so your custom classes automatically inherit some basic validation behavior.

The default implementations of these methods check the object's state against constraints and rules defined in your model.

There are two kinds of validation methods. The first validates an entire object to see if it's ready for a specific operation (inserting, updating, and deleting) and the second validates individual properties. The two different types are discussed in more detail in the following sections. For more a more detailed discussion, see the EOValidation interface specification in the *Enterprise Objects Framework Reference*.

Validating Before an Operation

The methods for validating an object before a specific operation are:

- validateForSave
- validateForDelete
- validateForInsert
- validateForUpdate

When you perform a particular operation on an enterprise object (such as attempting to delete it), the EOEditingContext sends a validation message to your enterprise object appropriate to the operation. Based on the result, the operation is either accepted or refused. For example, referential integrity constraints in your model might state that you can't delete a Customer object that has Rentals. If a user attempts to delete a customer that has rentals, the deletion is refused.

You might want to override these methods to perform additional validation to what can be specified in a model. For example, your application should never allow a fee to be deleted if it hasn't been paid.

You could implement this in a Fee class as follows:

In Java:

```
public void validateForDelete() throws
EOValidation.Exception
{
  EOValidation.Exception superException = null;
  EOValidation.Exception myException = null;

  try {
      super.validateForDelete()
  } catch (EOValidation.Exception s) {
      superException = s;
  }

  if (!isPaid())
      myException = new EOValidation.Exception(
          "You can't remove an unpaid fee.");

  if (superException && myException) {
      NSMutableArray exceptions = new NSMutableArray();
      exceptions.addObject(superException);
      exceptions.addObject(myException);
      throw

EOValidation.Exception.aggregateExceptionWithExceptions(
              exceptions);
  } else if (superException) {
      throw(superException);
  } else if (myException) {
      throw(myException);
  }
}
```

In Objective-C:

```
- (NSException *)validateForDelete
{
  NSException *superException = [super validateForSave];
  NSException *myException = nil;

  if (![self isPaid])
      myException = [NSException
validationExceptionWithFormat:
          @"You can't remove an unpaid fee."];

  if (superException && myException) {
      return [NSException
aggregateExceptionWithExceptions:
          [NSArray arrayWithObjects:
          superException, myException, nil];
  } else if (superException) {
      return superException;
  } else if (myException) {
      return myException)
  }

  return nil;
}
```

The default implementation of **validateForDelete** inherited by your custom classes performs basic checking based on referential integrity rules specified in your model. Your custom classes should therefore invoke super's implementation before performing their own validation as and should combine any exception thrown by super's implementation with their own as shown above.

Note that the Java and Objective-C validation methods work slightly differently. The Java methods throw exceptions to indicate a validation failure whereas the Objective-C methods create and return exceptions for the Framework to raise.

The advantage of using one of the **validateFor...** methods is that they allow you to perform both "inter-" and "intra-" object validation before a particular operation occurs. This is useful if you have an enterprise object whose properties have interdependencies. For example, if you're saving rental data, you might want to confirm that the check out date precedes the return date before committing the changes.

Validating Individual Properties

EOValidation defines a fifth method for validating individual properties: **validateValueForKey** (**validateValue:forKey:** in Objective-C). It's like the key-value coding methods in that it validates properties by name (or *key*). The default implementation of **validateValueForKey** that your custom classes inherit searches for a method of the form **validate***Key* and invokes it if it exists. These are the methods that your custom classes can implement to validate individual properties. For example, the Customer class might have a **validateStreetAddress** method:

In Java:

```
public void validateStreetAddress(String address)
  throws EOValidation.Exception
{
  if (/** address is invalid */)
      throw new EOValidation.Exception("Invalid
address.");
}
```

In Objective-C:

```
- (NSException *)validateStreetAddress:
      (NSString *)address
{
  if (/* address is invalid...*/)
      return [NSException validationExceptionWithFormat:
         @"Invalid address."];

  return nil;
}
```

If a user tries to assign a value to **streetAddress**, the default implementation of **validateValueForKey** invokes the **validateStreetAddress** method. Based on the result, the operation is either accepted or refused.

Prior to invoking your validateKey methods, the default implementation of **validateValueForKey** validates the property against constraints specified in your model (such as NULL constraints).

Note that the default implementations of **validateForSave**, **validateForInsert**, and **validateForUpdate** provided by EOCustomObject (NSObject in Objective-C) invoke **validateValueForKey** for each of an object's class properties.

Validating User Input

Besides putting validation in a model and in an enterprise object class, you can also put validation in the user interface. The way that validation is normally added to the user interface is through formatters, which perform data type and formatting validation when users enter values.

Additionally, you can design your Application Kit and Java client applications so that the validation in your enterprise objects is performed as soon as a user attempts to leave an editable field in the user interface. There are two steps to doing this:

1. In Interface Builder, display the Attributes view of the EODisplayGroup Inspector and check "Validate immediately".

2. In your enterprise object class, implement **validate***Property* to check the value of the key for which the user is entering a value. For example, if the key is **salary**, you'd implement the method **validateSalary**.

If **validateSalary** fails (that is, if the salary value isn't within acceptable bounds), the user is forced to correct the value before being allowed to leave the field.

Note that in this scenario, **validateValueForKey** is invoked with values directly from the user interface. When **validateValueForKey** is invoked from **validateForSave**, **validateForInsert**, or **validateForUpdate**, it's invoked with values that are already in the object.

Creating and Inserting Objects

In an Enterprise Objects Framework application, when new enterprise objects are inserted into the database, it's often through a display group—either an EODisplayGroup for Application Kit and Java client applications or a WODisplayGroup for WebObjects applications (assuming the application has a graphical user interface). However, it's also common to create and insert an enterprise object programmatically—either because your application doesn't have a graphical user interface, or because you're creating and inserting objects as the by-product of another operation.

To create an instance of a custom Java enterprise object class, you invoke a constructor of the following form:

```
public MyCustomEO (
  EOEditingContext anEOEditingContext,
  EOClassDescription anEOClassDescription,
  EOGlobalID anEOGlobalID)
```

Typically you invoke this constructor with **null** arguments as follows:

```
rental = new MyCustomEO(null, null, null);
```

You can provide real values, but most enterprise object classes don't make use of the arguments. The only common exception to this is EOGenericRecord, which requires the arguments to identify the object's class description. EOGenericRecord is handled in a different way, as described later in this section.

In Objective-C, you use **alloc** and **init** like you would with any other object, so you don't have to worry about providing arguments.

Once you create an object, you insert it into an EOEditingContext using EOEditingContext's **insertObject** method (**insertObject:** in Objective-C). For example, the following code excerpts for the Customer class create Fee and Rental enterprise objects as the by-product of a customer renting a unit at a video store. Once the objects have been created, they're inserted into the current enterprise object's EOEditingContext.

In Java:

```
public void rentUnit(Unit unit)
{
  EOEditingContext ec;
  Fee fee;
  Rental rental;

  ec = editingContext();

  // Create new objects and insert them into the editing
context
  fee = new Fee(null, null, null);
  ec.insertObject(fee);
  rental = new Rental(null, null, null);
  ec.insertObject(rental);

  // Manipulate relationships
  rental addObjectToBothSidesOfRelationshipWithKey(fee,
"fees");
  rental.addObjectToBothSidesOfRelationshipWithKey(unit,
"unit");
  addObjectToBothSidesOfRelationshipWithKey(rental,
"rentals");
}
```

In Objective-C:

```
- (void)rentUnit:(Unit *)unit
{
  EOEditingContext *ec;
  Fee *fee;
  Rental *rental;

  ec = [self editingContext];

  // Create new objects and insert them into the editing
context
  fee = [[[Fee alloc] init] autorelease];
  [ec insertObject:fee];
  rental = [[[Rental alloc] init] autorelease];
  [ec insertObject:rental];

  [rental addObject:fee
      toBothSidesOfRelationshipWithKey:@"fees"];
  [rental addObject:unit
      toBothSidesOfRelationshipWithKey:@"unit"];

  [self addObject:rental
      toBothSidesOfRelationshipWithKey:@"rentals"];
}
```

EOEditingContext's **insertObject** method has the effect of registering the specified enterprise object with the editing context. In other words, it registers the object to be inserted in the database the next time the editing context saves changes. Inserting a new object into an editing context also has the side effect of further initializing the object, so you should always insert new objects into an editing context before modifying them (such as by manipulating their relationships). For more information, see the section "Setting Defaults for New Enterprise Objects" on page 98.

You create an instances of EOGenericRecord differently from the way you create custom objects: To create a generic record, use EOClassDescription's **createInstanceWithEditingContext** method (**createInstanceWithEditingContext:globalID:zone:** in Objective-C) as follows:

In Java:

```
String entityName; // Assume this exists.
EOEditingContext editingContext; // Assume this exists.
EOClassDescription classDescription;
EOEnterpriseObject eo;

classDescription =
EOClassDescription.classDescriptionForEntityName(entityN
ame);

eo = classDescription.createInstanceWithEditingContext(
     editingContext, null);

if (eo) editingContext.insertObject(eo);
```

In Objective-C:

```
NSString *entityName; // Assume this exists.
EOEditingContext *editingContext; // Assume this exists.
EOClassDescription classDescription;
id eo;

classDescription = [EOClassDescription
     classDescriptionForEntityName:entityName];
eo = [classDescription
     createInstanceWithEditingContext:editingContext
     globalID:nil
     zone:[editingContext zone]];

if (eo) [editingContext insertObject:eo];
```

The EOClassDescription method **createInstanceWithEditingContext** is preferable to using an EOGenericRecord constructor (or to using EOGenericRecord's **init...** method in Objective-C) because the same code works if you later use a custom enterprise object class instead of EOGenericRecord. Strictly speaking, it's better to use EOClassDescription's **createInstanceWithEditingContext** for all types of enterprise objects—EOGenericRecords as well as your custom classes. The **createInstanceWithEditingContext** method is the most general way to create an enterprise object—it works regardless of whether the object is represented by a custom class or an EOGenericRecord.

Working with Relationships

In the code excerpt shown in the preceding section, notice that before the objects are inserted into the EOEditingContext, the method **addObjectToBothSidesOfRelationshipWithKey** (**addObject:toBothSidesOfRelationshipWithKey:** in Objective-C) is used. This method is part of the EORelationshipManipulation interface (informal protocol in Objective-C). EORelationshipManipulation provides methods for manipulating an enterprise object's relationship properties. It is a part of the EOEnterpriseObject interface, and so your custom classes inherit a default implementation from EOCustomObject (or NSObject in Objective-C).

addObjectToBothSidesOfRelationshipWithKey is used to manage reciprocal relationships, in which the destination entity of a relationship has a back reference to the source. For example, Fee and Unit both have back references to Rental, and Rental has a back reference to Customer. In other words, not only does the model define a relationship from Customer to Fee and Rental, it also defines a relationship from Rental back to Customer and from Fee to Rental. When you insert an object into a relationship (such as adding a new Rental object to Customer's **rentals** relationship property, which is an NSArray) *and* the inserted object has a back reference to the enterprise object, you need to be sure to add the object to both sides of the relationship. Otherwise, your object graph will get out of sync—your Customer object's **rentals** array will contain the Rental object, but the Rental object won't know the Customer who rented it.

You can update object references explicitly—that is, you can directly update the Rental object's **customer** property, which represents its relationship to the Customer object. But it's simpler to just use **addObjectToBothSidesOfRelationshipWithKey**. This method is safe to use regardless of whether the source relationship is to-one or to-many, whether the reciprocal relationship is to-one or to-many, or whether the relationship is reciprocal at all.

You should observe the following guidelines in working with relationships:

- For non-reciprocal relationships, you can just use your class's regular accessor methods. These methods have the form **addTo***Property* (**addTo***Property***:** in Objective-C) where *Property* is a relationship array.
- When in doubt, use the method **addObjectToBothSidesOfRelationshipWithKey**. This method works for reciprocal and non-reciprocal relationships alike, and it's the safest choice if you don't know ahead of time what objects your code will be working with—for example, if you're providing a framework to be used by others.
- If you're working with reciprocal relationships and you do know the objects you'll be working with ahead of time, the best choice is to implement custom accessor methods in your enterprise object class to handle the reciprocal relationships. Custom accessor methods are preferable to **addObjectToBothSidesOfRelationshipWithKey** simply because they provide type checking.

In addition to the **addObjectToBothSidesOfRelationshipWithKey** method, EORelationshipManipulation defines the following methods:

- addObjectToPropertyWithKey
- removeObjectFromPropertyWithKey
- removeObjectFromBothSidesOfRelationshipWithKey

The method **addObjectToPropertyWithKey** (**addObject:toPropertyWithKey:** in Objective-C) adds a specified object to the relationship property (an NSArray) with the specified name (key).

This method is the primitive used by **addObjectToBothSidesOfRelationshipWithKey**. The default implementation uses the class definition as follows:

1. If this method is passed the key **projects**, for example, the default implementation looks for a method with the name **addToProjects**.

2. If the **addToProjects** method isn't found, **addObjectToPropertyWithKey** looks for a property called **projects**. If it finds it, it adds the argument to the array (assuming the array is mutable). If the array is immutable, it creates a new version of the array that includes the new element.

3. If the property is **null** (**nil** in Objective-C), a new empty array is created and assigned to the object.

The method **removeObjectFromPropertyWithKey** (**removeObject:fromPropertyWithKey:** in Objective-C) removes a specified object from the relationship property (an NSArray) with the specified name (key).

This method follows the same pattern as **addObjectToPropertyWithKey**. That is, it looks for a selector of the form **removeFromProjects**, and then for a property called **projects**. If neither of these can be found, it throws an exception. If it finds the property but it doesn't contain the specified object, this method simply returns.

The method **removeObjectFromBothSidesOfRelationshipWithKey** (**removeObject:fromBothSidesOfRelationshipWithKey:** in Objective-C) removes a specified object from both sides of a relationship property with the specified name. Like **addObjectToBothSidesOfRelationshipWithKey**, this method is safe to use regardless of whether the source relationship is to-one or to-many, or whether the reciprocal relationship is to-one or to-many.

Setting Defaults for New Enterprise Objects

When new objects are created in your application and inserted into the database, it's common to assign default values to some of their properties. For example, the Member class has a **memberSince** property. It's likely that you would assign that property a value when you create and insert a new object instead of forcing the user to supply a value for it.

To assign default values to newly created enterprise objects, you use the method **awakeFromInsertionInEditingContext** (**awakeFromInsertionInEditingContext:** in Objective-C). This method is automatically invoked immediately after your enterprise object class creates a new object and inserts it into the EOEditingContext.

The following implementation of **awakeFromInsertion** in the Member class sets the current date as the value of the **memberSince** property:

In Java:

```
public void awakeFromInsertion(EOEditingContext ec)
{
  super.awakeFromInsertion(ec);
  // Assign current date to memberSince
  if (memberSince == null)
      memberSince = new NSGregorianDate();
}
```

In Objective-C:

```
- (void)
awakeFromInsertionInEditingContext:(EOEditingContext
*)ec
{
  [super awakeFromInsertionInEditingContext:ec];
  // Assign current date to memberSince
  if (!memberSince)
      memberSince = [[NSCalendarDate date] retain];
}
```

You use the **awakeFromInsertion** method to set default values for enterprise objects that represent new data. For enterprise objects that represent existing data, the Enterprise Objects Framework provides the method **awakeFromFetch** (**awakeFromFetchInEditingContext:** in Objective-C), which is sent to an enterprise object that has just been created from a database row and initialized with database values. Your enterprise objects can implement this method to provide additional initialization. Because the Framework is still busy fetching when an enterprise object receives **awakeFromFetch**, the object must be careful about sending messages to other enterprise objects that might need to be *faulted* in. For more information about faulting, see the chapter "Behind the Scenes" and the EOFaulting interface specification in the *Enterprise Objects Framework Reference*.

Initializing Enterprise Objects

Since all of an enterprise object's class property values are assigned through the key-value coding methods, no special initialization is usually needed in your constructors (or **init...** methods in Objective-C). Specifically, you don't generally assign default values to your enterprise objects in constructors since it is invoked to create instances of your class being fetched from the database. Any values you assign in a constructor will be overwritten by the values fetched from the database.

However you can take advantage of extra information available at the time your enterprise object is initialized. In Java, instances of custom classes must provide a constructor of the following form:

```
public MyCustomEO (
  EOEditingContext anEOEditingContext,
  EOClassDescription anEOClassDescription,
  EOGlobalID anEOGlobalID)
```

The Framework uses such a constructor to create your objects, so you can use the provided arguments to influence the creation. In Objective-C, enterprise objects can be created with either **init** or **initWithEditingContext:classDescription:globalID:**. If an enterprise object implements the **initWithEditingContext:classDescription:globalID:** method, the Framework uses this method instead of **init**, allowing the object to affect its creation based on the data provided.

Writing Business Logic

So far the examples in this chapter have focused on working with your enterprise objects in fairly simple ways: creating them, accessing their data, modifying them, and validating changes before you save them to the database. But enterprise object classes can also implement more sophisticated business logic. For example, suppose you need a pay method in your Fee class that sets the fee's **datePaid** property and also notifies the fee's rental that the fee has been paid. You could implement this as follows:

In Java:

```
public void pay()
{
  setDatePaid(new NSGregorianDate());

  // Notify the rental that this fee has been paid.
  rental.feePaid();
}
```

In Objective-C:

```
- (void)pay
{
  [self setDatePaid:[[NSCalendarDate calendarDate]
retain];

  // Notify the rental that this fee has been paid.
  [rental feePaid];
}
```

When you implement a method such as this in your enterprise object class, you don't have to be concerned with registering the changes or updating the values displayed in the user interface. An Enterprise Objects Framework application revolves around the graph of enterprise objects, and the Framework assumes responsibility for making sure that all parts of your application are notified when an object's value changes. Thus, all parts of your application have the same view of the data and remain in sync.

Gotchas

This section discusses some of the more subtle issues you need to be aware of in designing enterprise objects.

Constructor for Creating Enterprise Objects

Your Java custom enterprise object classes must provide a constructor of the following form:

```
public MyCustomEO (
  EOEditingContext anEOEditingContext,
  EOClassDescription anEOClassDescription,
  EOGlobalID anEOGlobalID)
```

Enterprise Objects Framework uses this constructor to create instances of your custom classes, and it throws an exception if it doesn't find such a constructor.

Numeric Values and NULL

An important issue to consider in using scalar types is that relational databases allow the use of a NULL value distinct from any numeric value. When a NULL value is encountered in **takeValueForKey** (**takeValue:forKey:** in Objective-C), your enterprise object will be passed **null** (**nil** in Objective-C). Since the scalar types can't accommodate **null**, the default implementations of the key-value coding methods throw an exception when asked to assign **null** to a scalar property. You should either design your database not to use NULL values for numeric columns, use Number to represent scalar values in your enterprise class (NSNumber in Objective-C), or implement the method **unableToSetNullForKey** (**unableToSetNilForKey:** in Objective-C) method in your enterprise object class.

unableToSetNullForKey is part of the EOKeyValueCoding interface, and you use it to set policy for an attempt to assign **null** to an instance variable that requires a scalar type. Classes can implement this method to determine the behavior when **null** is assigned to a property in an enterprise object that requires a scalar type (such as **int** or **double**). One possible implementation is to reinvoke **takeValueForKey** with a special constant value.

Cautions in Implementing Accessor Methods

Whether you're implementing accessor methods to be used by the key-value coding methods or overriding the key-value coding methods themselves, you need to be aware of a few issues. The first involves handling NULL values from the database; the second involves accessing property values while they're being set.

NULL values in a database come into the access layer as EONullValue objects (EONull objects in Objective-C), but they're converted to **null** (or **nil** in Objective-C) before they're passed to your enterprise objects. The preceding section described how this scheme can cause problems for properties with numeric types, but they can cause problems for other property types as well. If your database uses NULL values, your enterprise objects may want to check any object values received through accessor methods to see that they're not **null** (or **nil**) before sending them messages.

You can encounter another kind of problem if your object's accessor methods for one property assume that another property has already been set and exists in usable form. Enterprise Objects Framework doesn't guarantee the order that properties are set, so your object's accessor methods can't count on the values of other properties being initialized or usable. Also, when the Framework creates an enterprise object, it creates *faults* for related objects, and these faults can be passed to your enterprise objects in a key-value coding message while the Framework is busy fetching. Your accessor methods (or overridden key-value coding methods) should be doubly careful about sending messages to objects fetched through relationships, because these messages can cause a fault object to attempt a fetch while the Framework is already busy, resulting in resource contention.

In most cases, you can overcome this problem using the stored value methods described in the section "Private Access with the Stored Value Methods."

Don't Override equals

Your enterprise objects shouldn't override the **equals** method (**isEqual:** in Objective-C). This is because Enterprise Objects Framework relies on the default implementation to check instance equality rather than value equality.

Chapter 4

Advanced Enterprise Object Modeling

This chapter discusses advanced enterprise object modeling issues not covered in the chapter "Designing Enterprise Objects." It is organized into the following sections:

- "Modeling Complex Attributes" (page 107) describes how to work with RTF text, image data, and your own custom data types.
- "Modeling Relationships" (page 114) describes approaches for modeling optional to-one relationships, handling **null**-valued to-one relationships, and modeling many-to-many relationships.
- "Modeling Inheritance" (page 124) describes the three ways to model inheritance and the advantages and disadvantages of each.
- "Designing Database-Savvy Enterprise Objects" (page 135) describes how to implement an enterprise object class that knows how to fetch its instances.

Modeling Complex Attributes

When you create a new model, EOModeler provides a default mapping from external database data types to one of the primitive value classes:

Java	Objective-C
String	NSString
BigDecimal (java.math)	NSDecimalNumber
Number	NSNumber
NSGregorianDate	NSCalendarDate
NSData	NSData

For example, in Oracle, VARCHAR columns are mapped to Strings, DATE columns are mapped to NSGregorianDates, and so on.

The default mapping is appropriate for most data types, but you have to customize the mapping for attributes with special requirements.

For example, suppose you have a **photo** attribute that's stored in the database as a LONG RAW (an Oracle data type). When you create a new model, this attribute is mapped to an NSData. However, NSData is just an object-oriented wrapper for binary data—it doesn't have any methods for operating on images, which limits what you'd be able to do with the image in your application. So, for example, in an Application Kit application, you could customize the default behavior and map **photo** to an NSImage instead.

This section describes how to work with complex attributes such as RTF text, images, and custom data types. Of these attributes, RTF text is the easiest to handle. You don't have to change the default mapping at all, but you have to use special interface objects to display the RTF text.

For images and custom data types, you must customize the default mapping to map to a "non-primitive" class. To map to an existing class (such as NSImage for image attributes), you simply specify additional mapping information in EOModeler. You do the same work in EOModeler to use a custom class. However, you must also provide methods for translating instances of your class to instances of the primitive value classes.

RTF Text

RTF text is one type of data that is commonly stored in NSDatas. In the database, store RTF data in a binary data type such as Oracle's LONG RAW; and in your enterprise object, store it in an NSData instance variable. EOModeler automatically maps binary data types to NSData, so the default mapping is correct for RTF attributes.

Figure 21. Model Settings for an RTF Text Attribute

For Application Kit applications, Enterprise Objects Framework provides the EOTextAssociation class that extracts RTF text from an NSData object and displays it in an NSTextView object. To display an RTF text attribute in an Application Kit user interface, use Interface Builder to make a connection from a text object to your EODisplayGroup.

Note: Click in the text area to select the NSTextView before you control-drag to form the association. Unless the cursor is in the text area, control-dragging from the text area forms a connection from the text's super view, an NSScrollView. To be sure that the NSTextView is selected, open the inspector and check that it's displaying the NSTextView Inspector.

You probably wouldn't use RTF-formatted text in a WebObjects of Java client application. If you have RTF-formatted text that you want to use in a web application, the best approach is probably to filter the text to another format (such as plain text or HTML) before displaying it in a web page.

Images

You can store image data in a binary data type (for example, LONG RAW in Oracle or image in Sybase) in the database. Alternatively, you can store the path name to an image file in the file system. This second approach is often times more practical for web applications.

When you store image data in the database itself, Enterprise Objects Framework initially maps it to an NSData object. Depending on the type of application you're writing—WebObjects or Application Kit —you can leave the image as an NSData, or you can further map the NSData to an NSImage object.

Image data formats (EPS, BMP, and GIF, for example) are usually different for WebObjects and Application Kit applications. In a WebObjects application, you generally work with GIF and JPEG images; whereas Application Kit applications typically use EPS, TIFF, and BMP images.

There are numerous techniques for displaying images in a WebObjects application, so how you display them is up to you. (For more information, see the "*Dynamic Elements Reference*" in the WebObjects documentation set.) However, most of the approaches use binary image data, so it's customary to model image attributes as NSDatas when you're designing an enterprise object for use in a web application. If you store images in a binary column in the database, you can then simply use EOModeler's default mapping.

In an Application Kit application, model an enterprise object's image as an NSImage. In the EOModeler's Attribute Inspector:

1. Set the Internal Data Type pop-up list to Custom.
2. In the Class field, specify NSImage as the custom class.
3. In the Factory Method field, specify **imageWithData:** as the method to use to create NSImage instances.

4. In the Conversion Method field, specify **TIFFRepresentation** as the method to use to convert NSImages into a form that can be stored in the database.

5. Set the Init Argument pop-up list to NSData, indicating that the argument to the factory method (**imageWithData:**) is an NSData object.

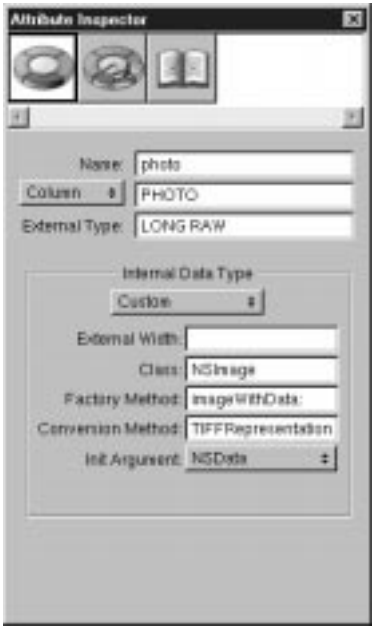


Figure 22. Model Settings for an Image Attribute in an Application Kit Application

To display an image attribute in an Application Kit user interface, make an EOControlAssociation from an NSImageView object to your EODisplayGroup.

Custom Data Types

In addition to supporting RTF text and image data, Enterprise Objects Framework also provides support for mapping attributes to custom objects. In other words, you can map an attribute to an Objective-C class that you have written—PhoneNumber, for example.

Enterprise Objects Framework maps attributes to a custom object using the same mechanism that it uses to map attributes to NSImages. In EOModeler's Attribute Inspector:

1. Set the Internal Data Type pop-up list to Custom.
2. If relevant, specify an external width in the External Width field.

   Binary data types such as Oracle's LONG RAW don't usually have width constraints. However, string columns such as VARCHARs often do have widths that you should enter.
3. In the Class field, specify the name of your custom class.
4. In the Factory Method field, specify the method to use to create instances of your class.

   This method should have one argument whose type matches the type specified in the Init Argument pop-up list.
5. In the Conversion Method field, specify the method to use to convert instances of your class into a form that can be stored in the database.

   This method should return an NSData object if the Init Argument type is NSData or Bytes, otherwise it should return an NSString.
6. Set the Init Argument pop-up list to indicate the data type (NSData, NSString, or Bytes) for the factory method's argument.

As an example, you might use the settings shown in Figure 23 for a PhoneNumber class.

Figure 23. Attribute Settings For a Custom Data Type

Using the mapping information in Figure 23, Enterprise Objects Framework fetches the **phoneNumber** attribute from the database, maps it to an NSString object, creates a PhoneNumber object using the PhoneNumber class method **phoneNumberWithString:**, and assigns the PhoneNumber object to its enterprise object. Similarly, Enterprise Objects Framework converts the PhoneNumber object to an NSString using the **phoneNumberAsString** method before saving it to the database.

Note: You can also use a Java custom value class. If you do, set the Factory Method to a static method in your custom class that creates instances. Also, the Init Argument must be set to either NSData or NSString (which is mapped to java.lang.String); it can't be set to bytes.

For more information on using custom data types, see the EOAttribute class specification in the *Enterprise Objects Framework Reference*.

Modeling Relationships

How you model relationships is perhaps the most complex and interesting part of a database-to-objects mapping. This section describes some of the finer points of relationship modeling.

Modeling Optional To-One Relationships

A to-one relationship is optional if the relationship's destination object can be **null** (**nil** in Objective-C). For example, a Member entity's **creditCard** relationship is optional if a Member object isn't required to have a CreditCard object.

Note: Occasionally, a mandatory to-one relationship doesn't resolve to a destination object. For example, suppose your application's Movie database contains legacy data for which relational integrity constraints weren't strictly enforced. As a result, some Movies don't have corresponding Studios even though the Movies-to-Studios relationship is mandatory. The techniques for handling these errant to-one relationships are the same as those for handling optional to-one relationships.

You can model an optional to-one relationship many different ways, depending on how you represent the relationship in the database—as a *foreign key to primary key join* or as a *primary key to primary key join*.

Note: For to-one relationships, Enterprise Objects Framework doesn't support primary key to foreign key joins. The destination join attribute in a to-one relationship must be the destination entity's primary key.

Using a foreign key to primary key join, you include the destination row's primary key in the source row. For example, in the relationship shown in Figure 24, the **creditCard** relationship's source table (MEMBER) has a foreign key (CARD_NUMBER) to the destination table (CREDIT_CARD). Using this approach, you can model an optional to-one relationship as a true to-one relationship just as you would model a mandatory to-one relationship.

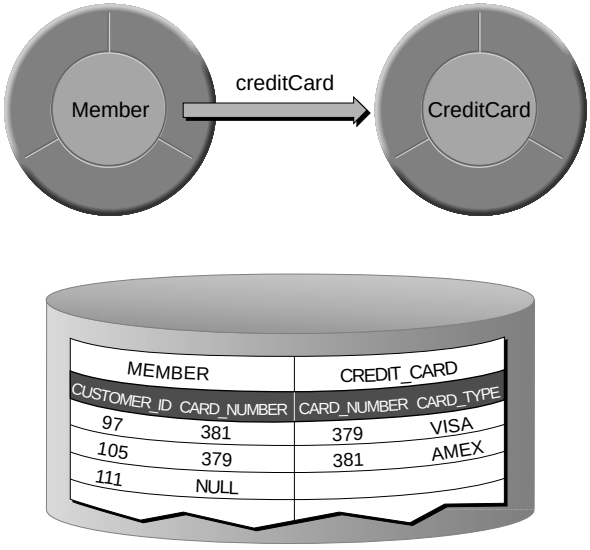


Figure 24. Storing a Foreign Key in the Source Table

A foreign key to primary key join is the best way to model a to-one relationship for use with Enterprise Objects Framework. If you have control over the design of the database schema, use foreign key to primary key joins for to-one relationships whenever possible.

Alternatively, you can use a primary key to primary key join that includes the source's primary key in the destination table. For example, in the relationship shown in Figure 25, the **talentPhoto** relationship's destination table (TALENT_PHOTO) has a foreign key (TALENT_ID) to the source table (TALENT).

For reasons described below, this arrangement requires special handling.

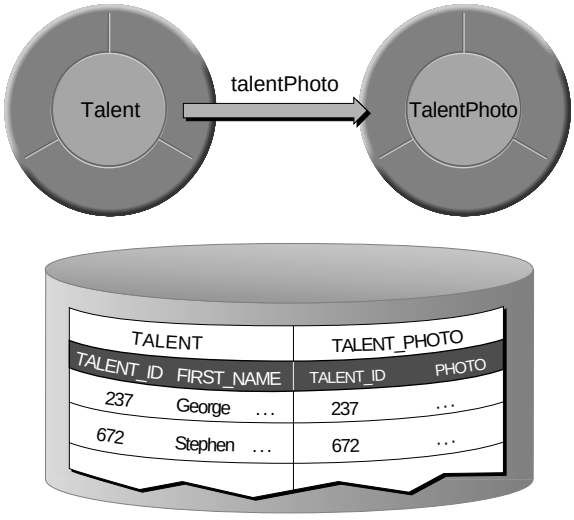


Figure 25. Storing a Foreign Key in the Destination Table

When Enterprise Objects Framework fetches an enterprise object, it attempts to assign destination objects to any of the object's to-one relationships. If a destination object hasn't already been fetched, Enterprise Objects Framework creates a fault to stand in for the destination object until it is actually needed. (For more information on faulting, see the chapter "Behind the Scenes" on page 187.)

The exception to this is when the relationship is based on a foreign key to primary key join and the relationship's source object doesn't have a corresponding destination. Instead of creating a fault, Enterprise Objects Framework assigns **null** (or **nil** in Objective-C) to the source object's relationship property. For example, if a Member doesn't have a corresponding CreditCard, the corresponding MEMBER record's CARD_NUMBER value is NULL. When Enterprise Objects Framework sees the null-valued CARD_NUMBER attribute, it sets the Member's **creditCard** property to **null**.

On the other hand, Enterprise Objects Framework can't detect that a primary key to primary key relationship doesn't have a destination. For example, Enterprise Objects Framework can't tell that a TALENT

record doesn't have a corresponding TALENT_PHOTO record until it tries to fetch a TALENT_PHOTO with the same TALENT_ID value and fails. Consequently, in a primary key to primary key relationship, Enterprise Objects Framework always assigns a fault if a corresponding destination object hasn't been fetched. If no such destination exists, Enterprise Objects Framework throws an exception when it tries to resolve the fault.

An optional primary key to primary key relationship (such as **talentPhoto**) can be handled in a number of ways:

- *Model the relationship as a mandatory to-one, but allow the destination entity to have* **null***-valued attributes.* For example, assume that a relationship between Customers and Addresses is optional. To model the relationship as a mandatory to-one, Customers who don't provide their Addresses have corresponding Address objects with **null**-valued **streetAddress**, **city**, **state**, and **zip** attributes. The Customers also have Address rows in the database, but the Address rows contain NULLs in all the columns except the primary key column (whose value matches that of the corresponding Customer).

  This approach is a good choice when the destination object contains what are conceptually attributes of the source object. For example, conceptually **photo** is an attribute of a Talent object. It's implemented using a to-one relationship for performance reasons. (Photo data is very large, and isn't fetched unless—and until—it's needed.)

- *Model the relationship as a to-many.* This approach is useful when you think that a to-one relationship may evolve into a to-many relationship in the future. For example, current requirements for a Movie application specify that a Talent object may only have one photo. However, the requirements for the next version of the application mention a Talent's portfolio.

- *Handle the exception thrown by faults that don't correspond to a destination object.* This approach is probably the best for handling optional to-one relationships based on primary key to primary key joins.

- I*mplement the delegate method* **databaseContextFailedToFetchObject** (**databaseContext:failedToFetchObject:globalID:** in Objective-C). This approach is best for handling mandatory to-one relationships with errant data (source rows that don't have corresponding destinations).

The following sections describe each approach.

Use a Mandatory To-One Relationship

This approach is used for the Movies database to model Talent's **talentPhoto** relationship. Although a Talent object doesn't have to have a photo, it does have to have a corresponding TalentPhoto object. As shown in Figure 26, a Talent object that doesn't have a photo has a TalentPhoto object whose **photo** attribute is **null** (**nil** in Objective-C).

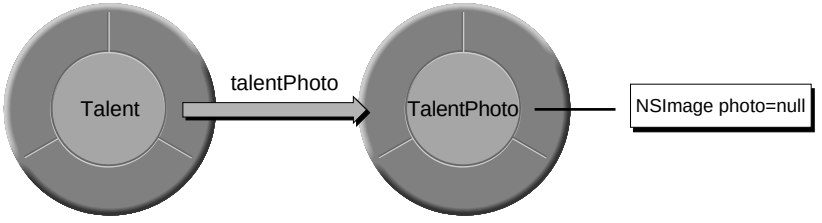


Figure 26. A Destination Object with **null**-Valued Attributes

This approach doesn't require any code. In the Advanced Relationship Inspector for the Talent entity's **talentPhoto** relationship, you simply set the relationship to *propagate primary key*. Propagate primary key tells Enterprise Objects Framework to propagate the primary key of the source entity into newly inserted objects in the destination entity (instead of generating a primary key value for the destination). With this configuration, Enterprise Objects Framework inserts a new Talent object, it inserts a corresponding TalentPhoto object if the Talent object doesn't already have one assigned to it.

Use a To-Many Relationship

To-many relationships use a different faulting mechanism than to-ones. A fault for a to-many relationship replaces itself with an NSArray of corresponding destination objects, and it doesn't throw an exception if it doesn't find any. If you use a to-many relationship to model an optional

to-one and no destination object exists, the array is simply empty. If the relationship does have a destination object, it's the first and only object in the array.

You can design your enterprise object's API to hide the to-many implementation. For example, suppose that Talent's **talentPhoto** relationship was modeled as a to-many. To design a Talent enterprise object that acts as if its **talentPhoto** relationship is an optional to-one, you could name the to-many relationship (and the corresponding instance variable) "_talentPhotoArray" and implement the following two accessor methods:

In Java:

```
public void setTalentPhoto(TalentPhoto talentPhoto)
{
  willChange();
  _talentPhotoArray.removeAllObjects();
  if (talentPhoto != null)
      _talentPhotoArray.addObject(talentPhoto);
}

public TalentPhoto talentPhoto()
{
  willRead();
  if (_talentPhotoArray.count() > 0)
      return _talentPhotoArray.objectAtIndex(0);
  return null;
}
```

In Objective-C:

```
- (void)setTalentPhoto:(TalentPhoto *)talentPhoto
{
  [self willChange];
  [_talentPhotoArray removeAllObjects];
  if (_talentPhotoArray)
      [_talentPhotoArray addObject:talentPhoto];
}

- (id)talentPhoto
{
  if ([_talentPhotoArray count])
      return [_talentPhotoArray objectAtIndex:0];
  return nil;
}
```

Handle the Exception

You can use a to-one relationship if you handle any exceptions that are thrown when a fault doesn't resolve to a destination object. For example, in the Talent enterprise object, you would implement the **talentPhoto** relationship "get" method as follows:

In Java:

```
public TalentPhoto talentPhoto()
{
  try {
      // If the receiver is a fault, sending it a willRead
      // message attempts to resolve it.  If the
      // corresponding row doesn't exist in the database,
      // an exception is thrown.
      talentPhoto.willRead();
  } catch (NSException e) {
      talentPhoto = null;
  }

  return talentPhoto;
}
```

In Objective-C:

```
- (TalentPhoto *)talentPhoto
{
  NS_DURING
      // If the receiver is a fault, sending it a self
      // message attempts to resolve it.  If the
      // corresponding row doesn't exist in the database,
      // an exception is raised.
      [talentPhoto self];
  NS_HANDLER
      [talentPhoto autorelease];
      talentPhoto = nil;
  NS_ENDHANDLER

  return talentPhoto;
}
```

Sending **willRead** (or **self** in Objective-C) to a fault triggers it to fetch its corresponding enterprise object. If a Talent instance doesn't have a corresponding TalentPhoto, sending **willRead** to the **talentPhoto** property throws an exception. In the **talentPhoto** method above, the exception handler simply sets the property to **null** (first autoreleasing the **talentPhoto** fault in Objective-C).

Implement databaseContextFailedToFetchObject

With the EODatabaseContext delegate method **databaseContextFailedToFetchObject** (**databaseContext:failedToFetchObject:globalID:** in Objective-C), you can prevent an exception from being thrown when a fault doesn't resolve to a destination object. This method is invoked when a fault for a to-one relationship can't find its corresponding object in the database. By returning **false** (NO in Objective-C), you can prevent the EODatabaseContext from raising an exception.

For example, to handle mandatory to-one relationships with errant data (source rows that don't have corresponding destinations), you could implement the delegate method to insert the empty object, thereby supplying the missing destination object:

In Java:

```
public boolean databaseContextFailedToFetchObject(
  EODatabaseContext context,
  Object object,
  EOGlobalID gid)
{
  // Perform a check to determine whether to intervene
  if (...) {
      // Set values in your object (if necesssary).
      object.editingContext().insertObject(object);
      return false;
  }
  return true;
}
```

In Objective-C:

```
- (BOOL)databaseContext:(EODatabaseContext *)context
  failedToFetchObject:(id)object
  globalID:(EOGlobalID *)gid
{
  // Perform a check to determine whether to intervene
  if (...) {
      // Set values in your object (if necesssary).
      [[object editingContext] insertObject:object];
      return NO;
  }
  return YES;
}
```

In the above implementations, the delegate method first checks to see if it should intervene. For example, the method might check to see if **object**

is an instance of the TalentPhoto class. If the delegate determines that **object** represents a destination object that's missing from the database, the delegate queues **object** for insertion into the database by inserting it into its editing context. It returns **false** (NO in Objective-C) indicating that the delegate has handled the error and that the EODatabaseContext shouldn't throw an exception.

Modeling Many-To-Many Relationships

To model a many-to-many relationship between objects is simple: each object manages a collection of the other kind. For example, consider the many-to-many relationship between employees and projects. To model this relationship in objects, an Employee has an NSArray, **projects**, of all the projects he or she works on; and a Project class has an NSArray, **employees**, of all its members.

To model a many-to-many relationship in a database, you have to create an intermediate table (also known as a *correlation* or *join* table). For example, the database for employees and projects might have EMPLOYEE, PROJECT, and EMP_PROJ tables, where EMP_PROJ is the correlation table. The appendix "Entity-Relationship Modeling" provides more information on the tables behind a many-to-many relationship.

Given the relational database representation of a many-to-many, how do you get the object model you want? You don't want to see evidence of the correlation table in your object model, and you don't want to write code to maintain database correlation rows. With Enterprise Objects Framework, you don't have to. Simply use flattened relationships as described in the chapter "Using EOModeler" to hide your correlation tables.

A model with the following features has the effect of hiding the EMP_PROJ correlation table from its object model altogether:

- Employee and Project entities whose to-many relationships to the EmpProj entity (**toEmpProj**) are not class properties.
- The flattened relationships **projects** and **employees** in Employee and Project, respectively, are class properties.

Consequently, EmpProj enterprise objects are never created, Employees have an array of related Projects, and Projects have an array of related Employees. Furthermore, Enterprise Objects Framework automatically manages rows in the EMP_PROJ correlation table.

However, what do you do when a correlation table contains extra attributes that are interesting? For example, the MOVIE_ROLE table in the sample Movies database is a correlation table between movies and the actors who star in them. In addition to foreign keys for MOVIE and TALENT, the MOVIE_ROLE table also contains the name of the role the actor plays in the film. In this case, MovieRole enterprise objects actually have a place in the object model even though they're fetched from a correlation table.

If you want to programmatically access both a Movie's roles and its actors directly from the Movie object, you should do the following:

1. Create a **movieRole** relationship from Movie to MovieRole and set it to be a class property.

2. Create a **talent** relationship from MovieRole to Talent.

3. Define an **actors** method in the Movie class that returns the Talent objects by getting them from the corresponding MovieRoles.

Because MovieRole corresponds to a data-bearing correlation table, you shouldn't create a flattened relationship from Movie to Talent. If your application fetches correlation records as enterprise objects, consistency problems can arise if it also manages a flattened relationship. For example, suppose you did flatten the **talent** relationship into the Movie entity. Movie objects would then have an array of MovieRole objects and an array of Talent objects. If your application adds a new MovieRole to a Movie's **roles** array, the corresponding **actors** array doesn't reflect the addition until the new MovieRole is saved to the database and the Movie is refetched.

Instead, if you create an **actors** method that traverses the object graph through the Movie's MovieRole objects, you avoid any consistency problems.

For display purposes, you don't even need an accessor method to bypass a correlation object. Instead, you can use key paths. For example, you can use the key path **roles.talent** to access a Movie's Talent objects in a master-detail configuration between a Movie EODisplayGroup and a Talent EODisplayGroup.

Modeling Inheritance

One of the issues that may arise in designing your enterprise objects—whether you're creating your schema from scratch or working with an existing database—is the modeling of inheritance relationships.

In object-oriented programming, when a subclass inherits from a superclass, the instantiation of the subclass implies that all the superclass' data is available for use by the subclass. When you instantiate objects of a subclass from database data, all database tables that contain the data held in each class (whether subclass or superclass) must be accessed so that the data can be retrieved and put in the appropriate enterprise objects.

Even in the simplest scenario in which there is a one-to-one mapping between a single database table and an enterprise object, the database and the enterprise objects instantiated from its data have no knowledge of each other. Their mapping is determined by an EOModel. Likewise, inheritance relationships between enterprise objects and the mapping of those relationships onto a database are also managed by an EOModel.

Note: Enterprise Objects Framework doesn't support mapping inheritance hierarchies across tables in separate databases. Instead, you can set up groups of objects connected by cross-database relationships, where related objects forward messages to each other.

Types of Inheritance

Suppose you're designing an application that includes Employee and Customer objects. Employees and customers share certain characteristics, such as a name and address, but they also have specialized characteristics. For example, an employee has a salary and a department, and a customer has account information.

Based on these data requirements, you might design a class hierarchy that has a Person class, and Employee and Customer subclasses. As subclasses of Person, Employee and Customer inherit Person's attributes (name and address), but they also implement attributes and behaviors that are specific to their classes.

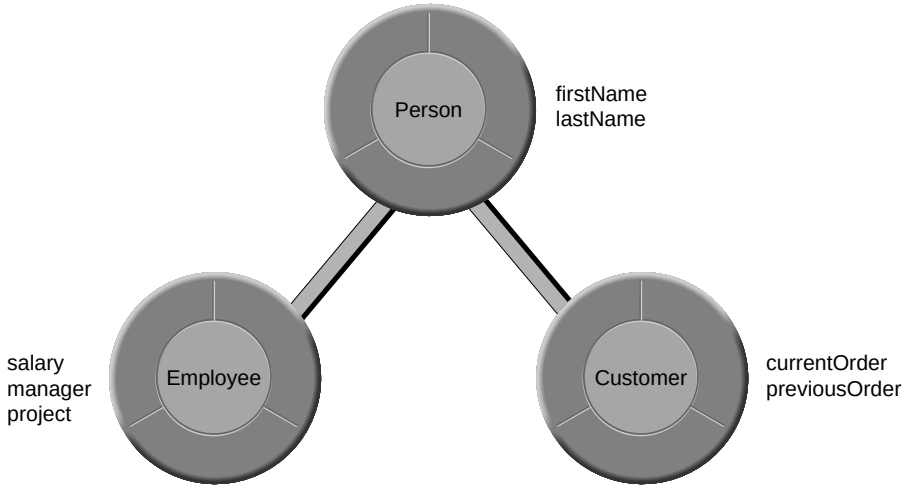


Figure 27. Class Hierarchy

In addition to designing your class hierarchy, you need to decide how to structure your database so that when objects of the classes are instantiated, the appropriate data is retrieved. Some of the issues you need to weigh in deciding on an approach are:

- Are fetches usually directed at the leaves or the root of the class hierarchy?

  When a class hierarchy is mapped onto a relational database, data is accessed in two different ways: By fetching just the leaves

(for example, just Employee or Customer), and by fetching at the root (Person) to get instances of all levels of the class hierarchy (Employees and Customers).

- How deep is the class hierarchy?

  While deep class hierarchies can be a useful technique in object-oriented programming, you should try to avoid them for enterprise objects. When you attempt to map a deep class hierarchy onto a relational database, the result is likely to be poor performance and a database that's difficult to maintain.

- What is the database storage cost for NULL attributes?
- Will I need to modify my schema on an ongoing basis?
- Will other tools be accessing the database?
- Do I need to use inheritance at all?

The primary consideration in deciding whether to use inheritance is if you'll ever need to perform a deep fetch. In an inheritance hierarchy, fetching just objects of a particular class is a shallow fetch, while fetching all instances of a class and its subclasses is a deep fetch. For example, even if the Objective-C classes for Customer and Employee inherit from Person, if your application never performs a fetch for "all people including Customers and Employees," then there is no need to tell Enterprise Objects Framework about your class hierarchy.

Enterprise Objects Framework supports the three primary approaches for mapping inheritance hierarchies to database tables:

- Vertical mapping
- Horizontal mapping
- One table mapping

These approaches, along with the advantages and disadvantages of each, are discussed in the following sections. None of them represents a perfect solution—which one is appropriate depends on the needs of your application.

Vertical Mapping

In this approach, each class has a separate table associated with it. There is a Person table, an Employee table, and a Customer table; each table contains only the attributes defined by that class.

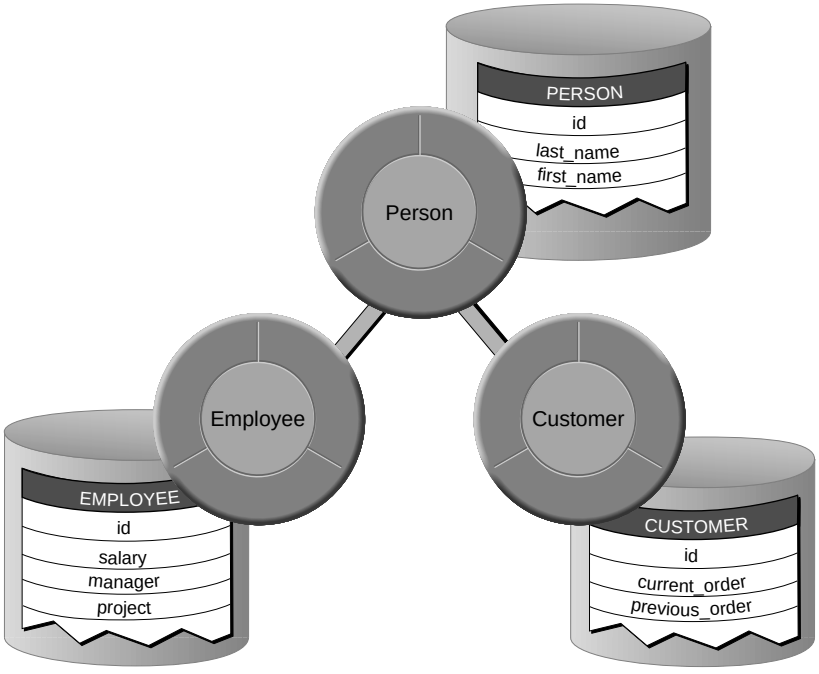


Figure 28. Vertical Inheritance Mapping

This method of storage directly reflects the class hierarchy. If an object of the Employee class is retrieved, data for the Employee's Person attributes must be fetched along with Employee data. The relationship between Employee and Person is resolved through a join to give Employee access to its Person data. This is also true for Customer.

Creating an EOModel for Vertical Mapping

Assuming that the entities for each of the participating tables already exist, you do the following to implement vertical mapping in your EOModel:

1. Create a to-one relationship from each of the child entities (Employee and Customer) to the parent entity (Person) joining on the primary keys, and set it so it isn't a class property.

2. Flatten the Person parent attributes into each child entity (Employee and Customer) setting them as class properties if they are class properties in Person.

3. Flatten the Person parent entity's relationships into each child entity (Employee and Customer), setting them as class properties if they are class properties in Person.

4. Set the parent entity for each child entity (Employee and Customer) to Person.

5. In the Advanced Entity Inspector, set the Person parent entity to be abstract if you won't ever instantiate instances of Person.

Advantages

With vertical mapping, a subclass can be added at any time without modifying the Person table. Existing subclasses can also be modified without affecting the other classes in the class hierarchy. The primary virtue of this approach is its clean, "normalized" design.

Disadvantages

Vertical mapping is the least efficient of all of the approaches. Every layer of the class hierarchy requires a join to resolve the relationships. For example, if you want to do a deep fetch from Person, three fetches are performed: a fetch from Employee (with a join to Person), a fetch from Customer (with a join to Person), and a fetch from Person to retrieve all the Person attributes. (If Person is an abstract superclass for which no objects are ever instantiated, the last fetch is not performed.)

Horizontal Mapping

In this approach, you have separate tables for Employee and Customer that each contain columns for Person. The Employee and Customer tables contain not only their own attributes, but all of the Person attributes as well. If instances of Person exist that are not classified as Employees or Customers, a third table would be required (where Person is *not* an abstract class). In other words, with horizontal mapping every concrete class has a self-contained database table that contains all of the attributes necessary to instantiate objects of the class.

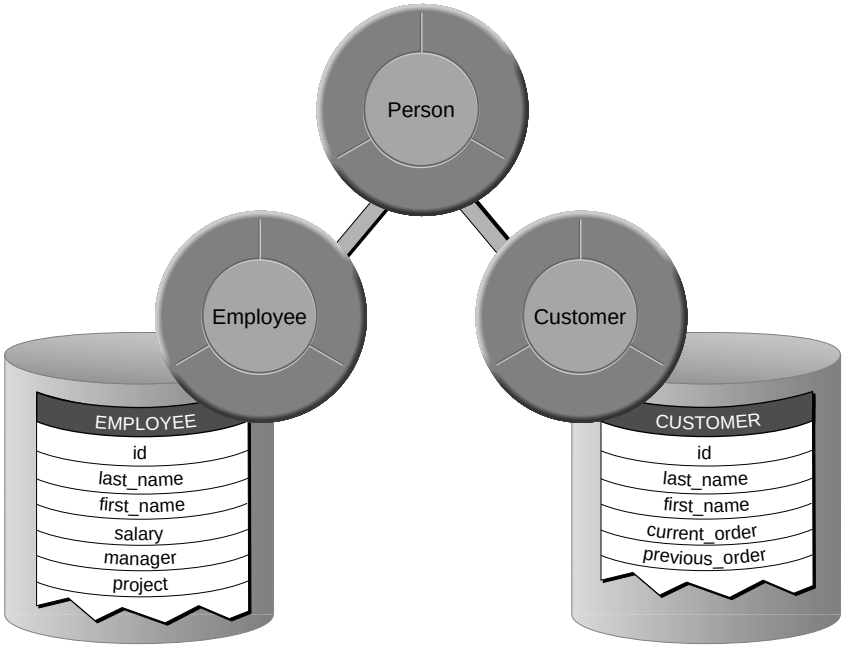


Figure 29. Horizontal Inheritance Mapping

This technique entails the same fetching pattern as vertical mapping, except that no joins are performed.

Creating an EOModel for Horizontal Mapping

To implement horizontal mapping, you do the following in your EOModel:

1. If a Person entity doesn't already exist, create one. (If there isn't a database table exclusively for Persons who aren't Employees or Customers, EOModeler doesn't automatically create an entity for Person.)

2. Set Person as the parent entity of Employee and Customer.

3. In the Advanced Entity Inspector, set the parent entity to be abstract if you never fetch Person objects that aren't Employees or Customers (you never instantiate instances of Person). Under horizontal mapping, if Person doesn't have its own table, then it's an abstract entity.

Unlike vertical mapping, you don't need to flatten any of Person's attributes into Employee and Customer since they already include all of its attributes.

Advantages

Similar to vertical mapping, a subclass can be added at any time without modifying other tables. Existing subclasses can also be modified without affecting the other classes in the class hierarchy.

This approach works well for deep class hierarchies, as long as the fetch occurs against the leaves of the class hierarchy (Employee and Customer) rather than against the root (Person). In the case of a deep fetch, it's more efficient than vertical mapping (since no joins are performed). It's the most efficient approach, if you only fetch instances of one leaf subclass at a time.

Disadvantages

Problems may occur when attributes need to be added to the Person superclass. The number of tables that need to be altered is equal to the number of subclasses—the more subclasses you have, the more effort is required to maintain the superclass. However, if table maintenance happens far less often than fetches, this might be a viable approach for your application.

Single Table Mapping

In this approach, you put all of the data in one table that contains all superclass and subclass attributes. Each row contains all of the columns for the superclass as well as for all of the subclasses. The attributes that don't apply for each object have NULL values. You fetch an Employee or Customer by using a query that just returns objects of the specified type (the table would probably include a type column to distinguish records of one type from the other).

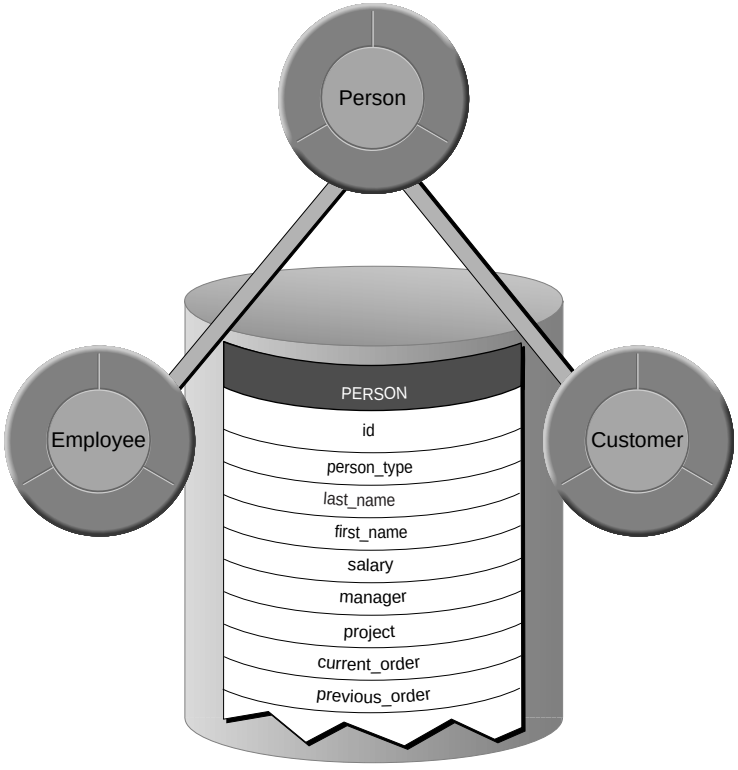


Figure 30. Single Table Mapping

Creating an EOModel for Single Table Mapping

To implement a single table mapping, you do the following in your EOModel:

1. Add Employee and Customer entities to your model. (You can use EOModeler's Create Subclass command for this purpose.)

2. Set Person as the parent entity of Employee and Customer.

3. In the Advanced Entity Inspector, assign a restricting qualifier to the Employee entity that distinguishes its rows from the rows of other entities. Similarly, assign a restricting qualifier to the Customer entity.

4. In the Advanced Entity Inspector, set the Person entity to be abstract if you won't ever instantiate Person instances.

Unlike vertical mapping, you don't need to flatten any of Person's attributes into Employee and Customer since these entities already have all of Person's attributes.

Each sub-entity maps to the same table and contains attributes only for the properties that are relevant for that class.

Advantages

This approach is faster than the other two methods for deep fetches. Unlike vertical or horizontal mapping, you can retrieve superclass objects with a single fetch, without performing joins. Adding a subclass or modifying the superclass requires changes to just one table.

Disadvantages

Single table mapping can consume an inordinate amount of space since every row includes columns for every one of the other entities' attributes. This may depend on how your database stores NULLs. Some databases condense NULL values, thereby reducing the storage space needed, but some databases maintain the length of the actual data type of the column regardless of the value stored. Most databases also have limitations on how many columns a table can have (typically this is around 250 columns), which can make it impossible to use single table mapping for a deep class hierarchy that has lots of instance variables.

Also, if you have a lot of data, this approach can actually be less efficient than horizontal mapping since with single table mapping you have to search the entire table to find the rows needed to instantiate objects of a particular type. (Horizontal mapping is only more efficient if your application just fetches one type of leaf object at a time (instances of a particular subclass).

Data Access Patterns for Inheritance

The following table summarizes how data is fetched in each of the approaches.

	Fetches from Leaves	Fetches from Root
Vertical Mapping	1 fetch using join	n fetches using join
Horizontal Mapping	1 fetch	n fetches
Single Table Mapping	1 fetch	1 fetch

In the table, "n" represents the number of entities involved in a deep fetch. For example, when you perform a deep fetch against Person in the Person, Customer, Employee class hierarchy, n equals 3.

Fetching and Inheritance

Once you've designed your class hierarchy and set up your EOModel to support that class hierarchy, you can use this information to fetch objects of the desired type. For example, you might want to just fetch Person objects, not Customer or Employee objects—or you might want to fetch all Person objects, including Customers and Employees.

You can control deep versus shallow fetches by using EOFetchSpecification's **setIsDeep** method (**setIsDeep:** in Objective-C). This method specifies whether a fetch should include sub-entities of the fetch specification's entity. If this method is set to **true** (YES in Objective-C), sub-entities are also fetched; if it's set to **false** (NO), they aren't. EOFetchSpecifications are deep by default.

When multiple entities are mapped to a single database table, you must set a qualifier on each entity to distinguish its rows from the rows of other entities. You can either do this programmatically by using EOEntity

method **setRestrictingQualifier** (**setRestrictingQualifier:** in Objective-C), or you can directly specify the qualifier in the EOModeler Advanced Entity Inspector. A restricting qualifier maps an entity to a subset of rows in a table. If you're using single table inheritance mapping, you must use a restricting qualifier that identifies objects of the desired type.

Delegation Hooks for Optimizing Inheritance

EOModelGroup includes delegate methods that you can use to exercise more fine-grained control over inheritance. These include:

relationshipForRow

This method (**entity:relationshipForRow:relationship:** in Objective-C) is invoked when relationships are instantiated for a newly fetched object. The delegate can use the information in the row to determine which entity the target enterprise object should be associated with, and replace the relationship appropriately.

subEntityForEntity

This method (**subEntityForEntity:primaryKey:isFinal:** in Objective-C) allows the delegate to fine-tune inheritance by indicating from which sub-entity an object should be fetched based on its primary key. The entity returned must be a sub-entity of the specified entity.

In Objective-C, if the delegate knows that the object should be fetched from the returned entity and not one of its sub-entities, it should set the **isFinal** argument (a pointer to a BOOL) to YES. (In Java, the object must be fetched from the returned entity; it must be final.)

classForObjectWithGlobalID

This method (**entity:classForObjectWithGlobalID:** in Objective-C) is also used to fine-tune inheritance. The delegate can use the specified globalID to determine a subclass to be used in place of the one specified in the entity argument.

Java Limitation With Ambiguous To-One Relationships

In both Java and Objective-C you can have to-many relationships to instances of both leaf and non-leaf subclasses in your class hierarchy. For example, the SoftballTeam entity can have a to-many relationship to Person or just the Employee entity (as in a company only team).

Similarly, you can have to-one relationships to instances of leaf subclasses. For example, the PurchaseOrder entity can have a to-one relationship to the Customer entity.

However, in Java you can *not* have a to-one relationship to a non-leaf entity, an *ambiguous to-one relationship*, unless you implement a workaround. Ambiguous to-one relationships are not possible in Java because of strong typing in the language. In Objective-C, the class of an instance can be changed after it is instantiated (i.e., from the parent class to the leaf class).

There are two workarounds for Java programmers. You can encode class information in your primary and foreign keys, and implement EOModelGroup's delegate methods as described in "Delegation Hooks for Optimizing Inheritance" on page 134. If you choose this option, then consider using single table mapping since the table would already contain class information.

The second workaround is to implement the ambiguous to-one relationship as a to-many, similar to dealing with an optional to-one relationship above. "Use a To-Many Relationship" on page 118

Designing Database-Savvy Enterprise Objects

The main function of a class object is to act as a factory for its instances, so a class method is a natural place to implement custom instantiation. When instances are initialized with data fetched from a database, why not implement common database queries in the class object as creation methods?

For example, a video rental store application based on the sample Rentals database might have a requirement to fetch all overdue rentals. To fulfill this requirement, you could write a Rental class method such as the following that fetches all its overdue instances:

In Java:

```
public static NSArray overdueRentalsWithEditingContext(
  EOEditingContext ec)
{
  NSGregorianDate today = new NSGregorianDate();
  EOQualifier qualifer = new EOKeyValueQualifier(
      "dueDate",
      EOQualifier.QualifierOperatorLessThan,
      today);
  EOFetchSpecification fetchSpec = new
EOFetchSpecification(
      @"Rental",
      qualifer,
      null);

  return ec.objectsWithFetchSpecification(fetchSpec);
}
```

In Objective-C:

```
+ (NSArray *)
  overdueRentalsWithEditingContext:
      (EOEditingContext *)ec
{
  NSCalendarDate *today = [NSCalendarDate calendarDate];
  EOQualifier *qualifer =
      [EOQualifier qualifierWithQualifierFormat:
          @"dueDate < %@",
          today];
  EOFetchSpecification *fetchSpec =
      [EOFetchSpecification
          fetchSpecificationWithEntityName:@"Rental"
          qualifier:qualifer sortOrderings:nil];

  return [ec objectsWithFetchSpecification:fetchSpec];
}
```

You would then invoke the method as follows:

In Java:

```
rentalArray =
Rental.overdueRentalsWithEditingContext(ec);
```

In Objective-C:

```
rentalArray = [Rental
overdueRentalsWithEditingContext:ec];
```

Note that **overdueRentalsWithEditingContext** takes an EOEditingContext as an argument. This is because enterprise objects are fetched into a particular EOEditingContext. An EOEditingContext establishes a single, internally consistent "object view" of the database, and an enterprise object in one editing context shouldn't have references to enterprise objects in another one. Consequently, methods that fetch enterprise objects must fetch them using the correct EOEditingContext.

Because a class object is global, its static methods (class methods in Objective-C) can be invoked from anywhere in an application. Thus, a class method that returns enterprise objects fetched from the database needs to receive the correct EOEditingContext as an argument.

Note: While the Rental class in this example closely resembles the Rental class for the sample Rentals database, **dueDate** has been added here to simplify the qualifier for fetching overdue rentals. The Rental class for the sample Rentals database doesn't have a **dueDate** attribute.

Part III

Application Design

Chapter 5

Application Configurations

When you use WebObjects Builder or Interface Builder to create the user interface of an Enterprise Objects Framework application, you also set up a network of behind-the-scenes objects including EODatabaseDataSources and EOEditingContexts. Some of these objects don't have representations in the builder applications, so you might not know they're there. Yet when you build and run your application, they're created automatically and they perform many important functions.

You rarely need to intervene in the automatic creation of these behind-the-scenes objects, but sometimes you need to interact with them. Therefore, it's important to know when they're created and ready to do work. The same is true in applications that don't have a graphical user interface.

The flexibility of Enterprise Objects Framework allows endless configurations, but most are variations on a few basic arrangements. This chapter explains how the plumbing for typical Enterprise Objects Framework applications is established and how to implement variations on the typical configurations. It's organized into the following major sections:

Graphical User Interface Applications

In a typical Enterprise Objects Framework application, a display group binds data fetched from a database to elements of the application's user interface. As shown in Figure 31, a display group relies on a network of objects to connect to a database and interact with it.

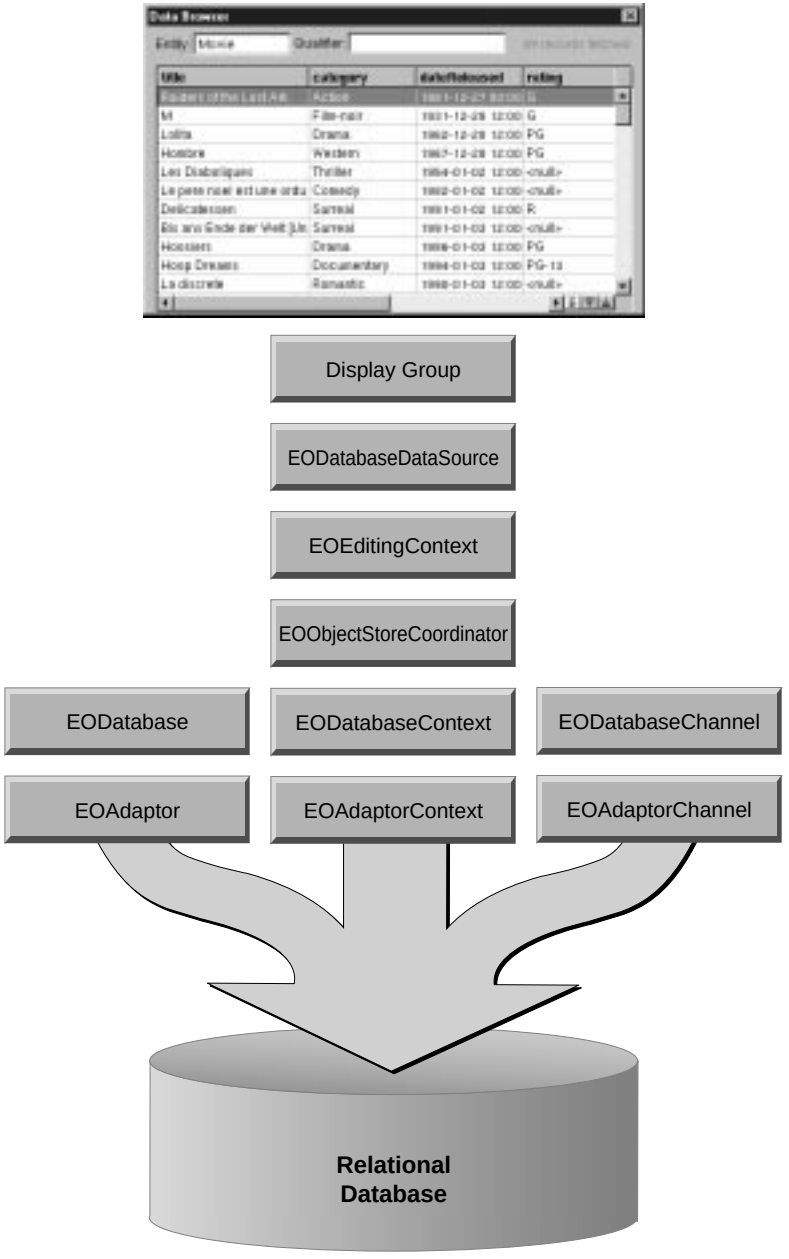


Figure 31. Typical Configuration of a Graphical User Interface Application

The network of objects is created automatically. For example, all the following procedures create a display group that is automatically associated with an EODatabaseDataSource and its underpinnings:

- Running the EOF Wizard in Project Builder (to create an Application Kit Framework application with a graphical user interface).
- Dragging an entity from a model file into a nib file in Interface Builder.
- Running the WebObjects Application Wizard in Project Builder to create a Main component with database access.
- Dragging an entity from a model file into a component in WebObjects Builder.
- Creating a "Direct-to-Web" application.

For more information on these classes and their roles in a database application, see the chapter "Enterprise Objects Framework Viewed Through Its Classes" on page 43 and the class specification for each in the *Enterprise Objects Framework Reference*.

The following sections describe how the network of objects is created, starting with a nib or component.

Loading a User Interface

When you drag an entity from a model file into Interface Builder or WebObjects Builder, you create an *entity display group*—a display group that's connected to an EODatabaseDataSource. The builder application automatically creates a display group, an EODatabaseDataSource, and an EOEditingContext. These objects are *archived* in your nib file or WebObjects component directory. At run-time, your application *unarchives* these objects as their interface is loaded, as shown in Figure 32. As part of unarchiving a nib file or component, many other objects are brought into the network of behind-the-scenes objects.

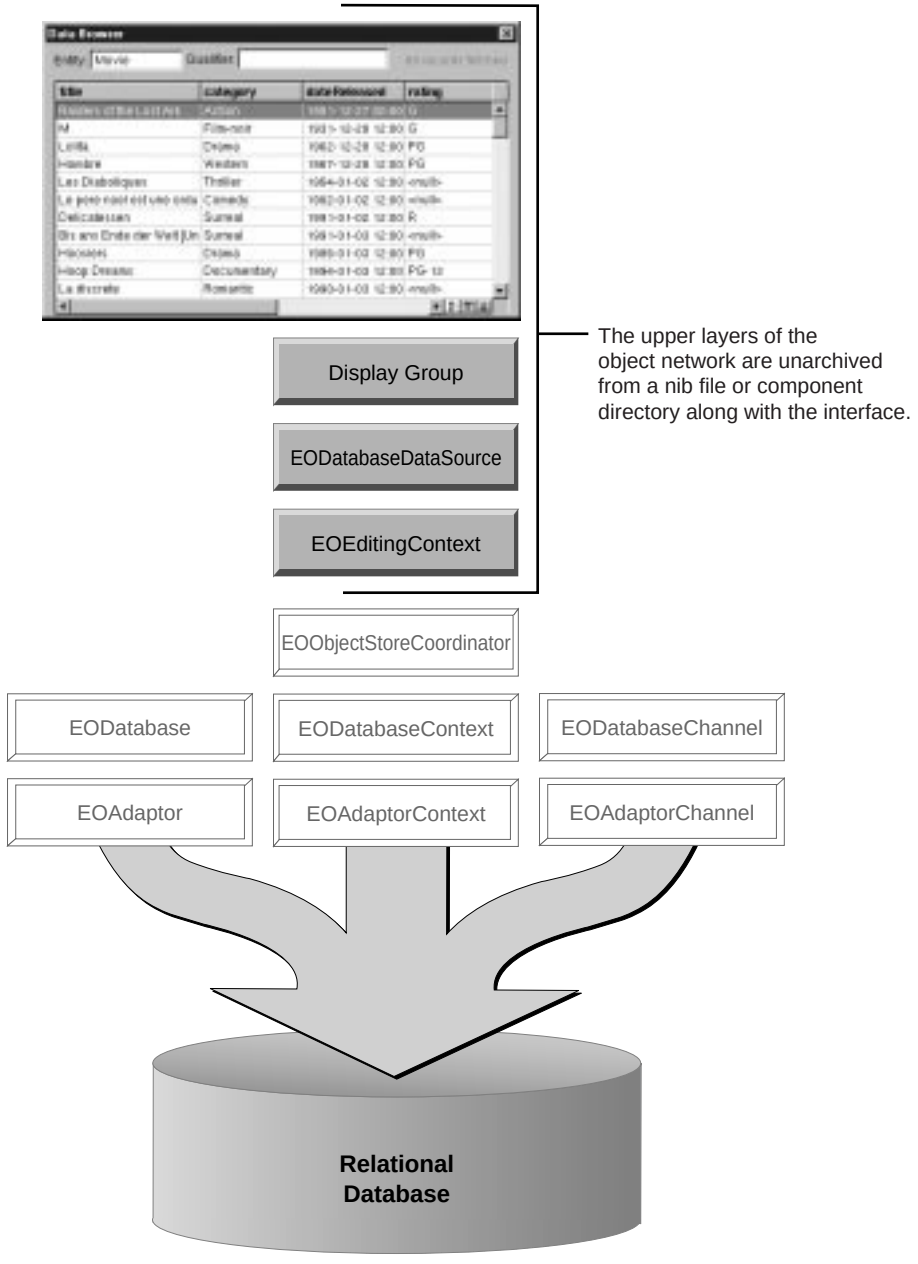


Figure 32. Enterprise Objects Framework Objects in a Nib or Component

Unarchiving an Editing Context

During unarchiving, an EOEditingContext uses the EOEditingContext class method **defaultParentObjectStore** to determine its parent object store. Normally, it's the EOObjectStoreCoordinator returned from the EOObjectStoreCoordinator class method **defaultCoordinator**. If a coordinator has not yet been created, it is created at this time.

Unarchiving a Database Data Source

Unarchiving an EODatabaseDataSource sets a more complex chain of events into motion: an EODatabaseContext and a host of associated objects are brought into the network of objects as follows:

1. The EODatabaseDataSource verifies that its underlying EOObjectStoreCoordinator has an EOModelGroup that can service its entity. If you haven't assigned a model group to the coordinator, then the coordinator uses the default model group—the shared EOModelGroup instance returned by the EOModelGroup class method **defaultGroup**.

   The default model group is created the first time **defaultGroup** is invoked. Subsequent invocations return the same shared instance. It contains all the models for an application, as well as for any frameworks the application references. In the majority of applications, the default model group is sufficient. However, if your particular application requires different model grouping semantics, you can create your own EOModelGroup instance, add the appropriate models, and assign it to your application's EOObjectStoreCoordinator using the EOModelGroup method **setModelGroup** (or the EOObjectStoreCoordinator method **setModelGroup:** in Objective-C).

2. After establishing that its entity can be serviced, the EODatabaseDataSource connects to an EODatabaseContext using the EODatabaseContext static method **registeredDatabaseContextForModel** (**registeredDatabaseContextForModel:editingContext:** in Objective-C). This method checks to see if the application's EOObjectStoreCoordinator already has a database context that the data source can use. If it does, it returns that database context.

Otherwise it instantiates a new database context, adds it to the coordinator, and returns it.

When an EODatabaseDataSource connects to an EODatabaseContext, the database context brings in additional supporting objects. A database context can't exist without an EODatabase and an EOAdaptorContext. Similarly, an EODatabase and an EOAdaptorContext can't exist without an EOAdaptor. Thus, as shown in Figure 33, connecting to a database context also connects an EODatabase, an EOAdaptor, and an EOAdaptorContext. Furthermore, if the adaptor bundle associated with the EODatabaseDataSource's model hasn't yet been loaded, it is loaded now.

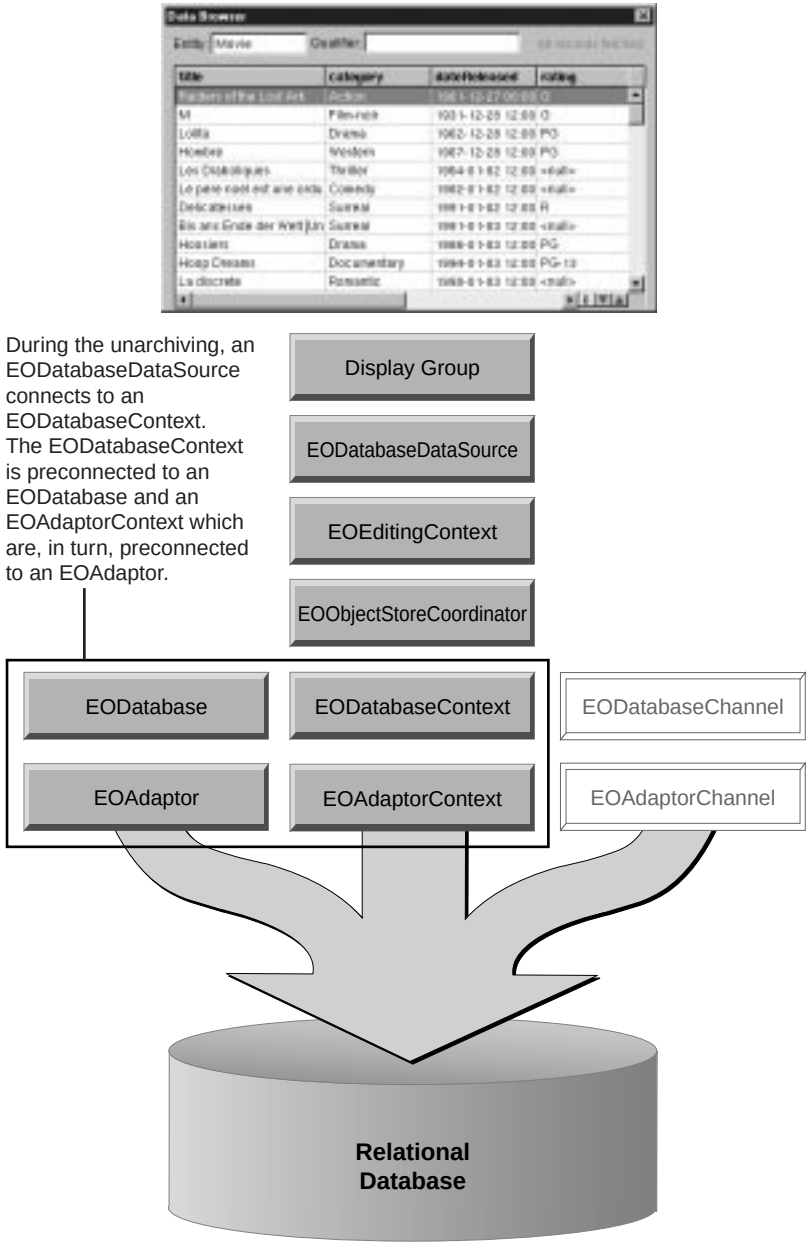


Figure 33. Connecting an EODatabaseDataSource with an EODatabaseContext

Sharing Editing Contexts and Coordinators

For Application Kit applications, Interface Builder creates only one EOEditingContext per nib. Therefore two entity display groups in the same nib share the same editing context by default.

Similarly, all editing contexts in an application share the same EOObjectStoreCoordinator by default.

If you want a different configuration, see the section "Editing Context Configurations" on page 159. It describes how objects in different nibs can share the same editing context and how to set up nested editing context. The section "Object Store Coordinator Configurations" on page 165 describes how you can set up multiple EOObjectStoreCoordinators.

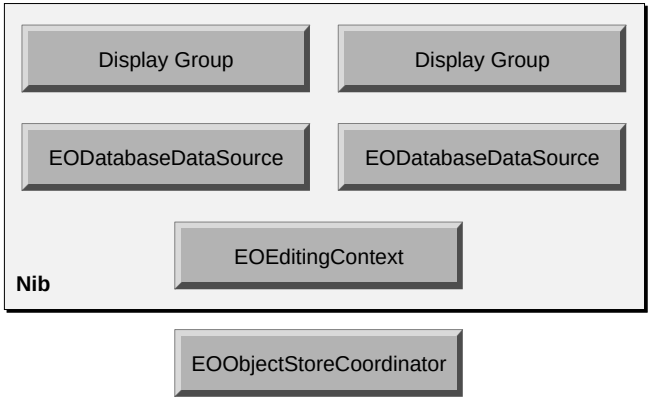


Figure 34. Display Groups in the Same Nib Share an Editing Context

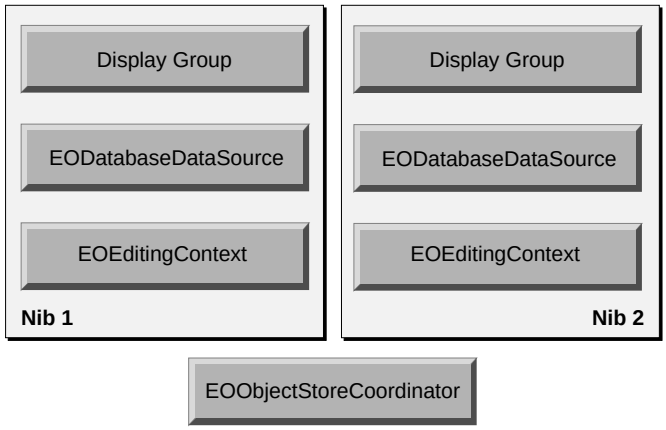


Figure 35. Display Groups in Different Nibs Have Separate Editing Contexts

Database Context Rendezvousing

To minimize the number of database connections an Enterprise Objects Framework application uses, an EODatabaseDataSource may share an existing EODatabaseContext with other data sources.

Using the EODatabaseContext static method **registeredDatabaseContextForModel** (**registeredDatabaseContextForModel:editingContext:** in Objective-C), a database data source *rendezvouses* with compatible database contexts whenever possible. A data source is compatible with a database context when the data source's model is compatible with the models in the EOModelGroup of the database context's EODatabase. Two models are compatible when their connection dictionaries are equal as determined by NSDictionary's **isEqualToDictionary** method (**isEqualToDictionary:** in Objective-C).

For example, in Figure 36, like-colored gray objects are associated with compatible models—models that have the same connection dictionary. The light gray data source is associated with the light gray database context, and the dark gray data sources are associated with the dark gray database context. The second dark gray data source to be unarchived rendevouses with the first data source's database context.

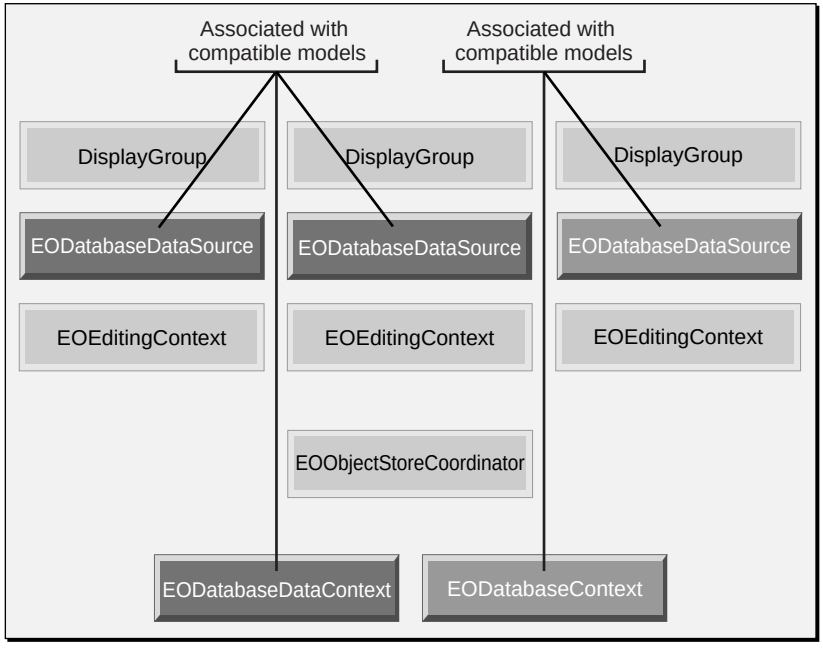


Figure 36. Data Sources Rendezvous With Compatible EODatabaseContexts

Note that two database data sources can be associated with different models and still share database contexts. So long as the models are compatible, they can be serviced by the same EODatabase and EODatabaseContext.

If you want to prevent an EODatabaseDataSource from rendezvousing on an existing EODatabaseContext, see "Object Store Coordinator Configurations" on page 165.

Setting Up Channels

The remaining objects in the network—an EODatabaseChannel and an EOAdaptorChannel—are created on demand when the application initiates a database interaction. For more information, see the section "Inside EODatabaseContext" on page 157.

Non-Graphical User Interface Applications

Command line tools, background processes, and other non-graphical user interface applications have a very similar configuration to that of applications with an interface. However, they don't use display groups, and they typically don't use EODatabaseDataSources either.

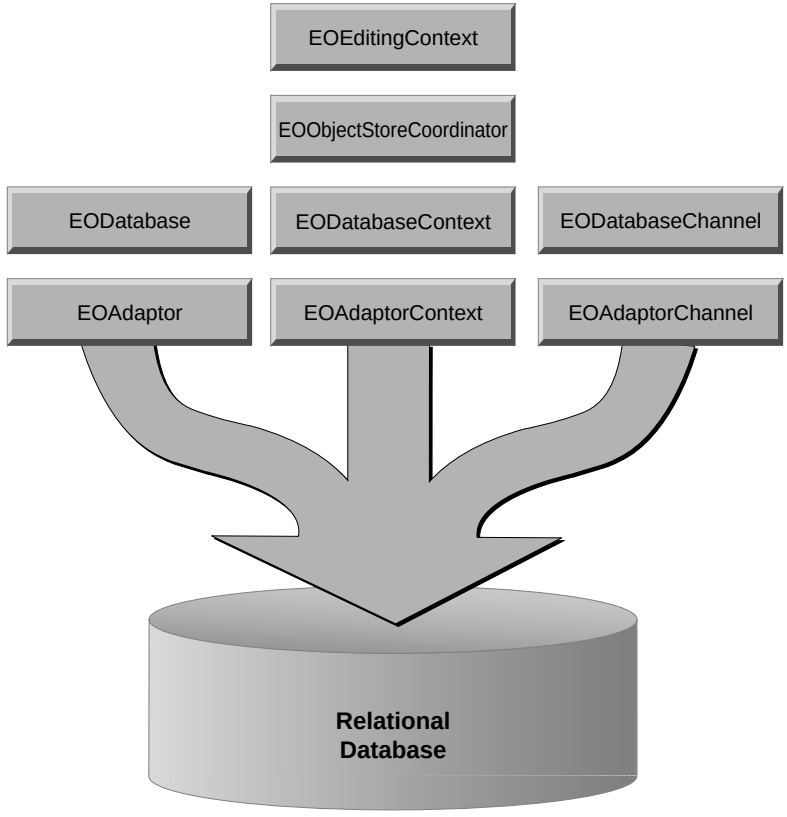


Figure 37. Typical Configuration of a Non-Graphical-User-Interface Application

In an application that doesn't have a graphical user interface, your code must initiate the creation of the network of the behind-the-scenes objects. This creation process is typically begun when you allocate and initialize an EOEditingContext.

Creating an Editing Context

You create an EOEditingContext the same way you'd create any other object:

In Java:

```
EOEditingContext editingContext = new EOEditingContext();
```

In Objective-C:

```
EOEditingContext *editingContext =
     [[EOEditingContext alloc] init];
```

Both of the examples above create a new editing context and connect it to an EOObjectStoreCoordinator. By default, the editing context is connected to the default object store coordinator as determined by EOObjectStoreCoordinator's **defaultCoordinator** static method (class method in Objective-C).

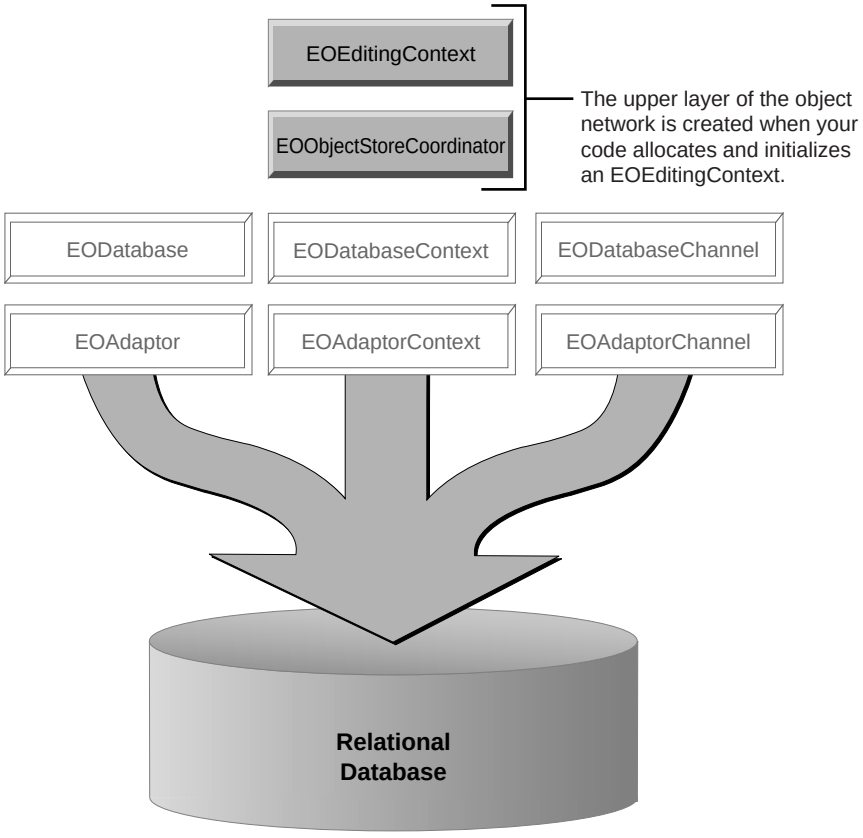
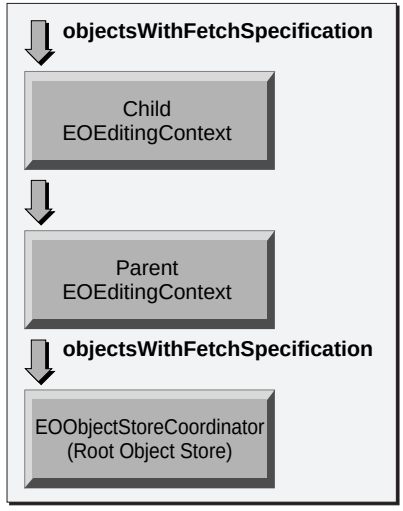


Figure 38. Allocating and Initializing an EOEditingContext

The first time **defaultCoordinator** is invoked, it creates an EOObjectStoreCoordinator. Subsequent invocations return the same instance. Consequently, all the editing contexts in an application are connected to EODatabaseContext objects through the same EOObjectStoreCoordinator by default.

The remaining objects in the network are created on demand. When a database operation is initiated with a message to an editing context, the request is passed on to its object store coordinator. In the case of a nested editing context configuration, the message is passed down the object network until reaches the editing context's root object store—usually an EOObjectStoreCoordinator.

For example, an **objectsWithFetchSpecification** message sent to an EOEditingContext percolates through its parent object stores until it reaches the coordinator as shown in Figure 39.



In a nested EOEditingContext configuration the child editing context relays an objectsWithFetchSpecification message to its parent object store, which relays the message to its own parent object store, and so on until the message reaches the root object store (usually an EOObjectStoreCoordinator).

Figure 39. How Messages Percolate Down to an EOObjectStoreCoordinator

Inside EOObjectStoreCoordinator

When an EOObjectStoreCoordinator receives a message requiring interaction with the database, it attempts to locate an EOCooperatingObjectStore—usually an EODatabaseContext—that can handle the request. The coordinator generally builds its list of cooperating stores on demand as follows:

1. When a coordinator needs to forward a request to a database, it checks to see if it has an cooperating stores in its list that can handle the interaction. If it finds one, it uses it.

2. If the coordinator does not have a cooperating store to handle the interaction (either because none of its cooperating stores can handle this specific database request or because the coordinator doesn't have any cooperating stores), it posts a CooperatingObjectStoreNeeded notification (EOCooperatingObjectStoreNeeded in Objective-C) so that another object can register a new cooperating store.

After posting the notification, the coordinator checks its list of cooperating stores a second time. If it finds an available store (because an object registered a new one in response to the notification), it uses the newly registered store.

In a typical Enterprise Objects Framework application, an EOObjectStoreCoordinator's cooperating stores are EODatabaseContexts. The EODatabaseContext class registers for the CooperatingObjectStoreNeeded notification, and provides the coordinator with a new database context that can accommodate the request. Consequently, you don't have to provide cooperating stores to a coordinator yourself unless you're using a subclass of EOCooperatingObjectStore that isn't an EODatabaseContext.

3. Once the coordinator has a cooperating store to use, it forwards the request to the store.

Note: In the case of an application with a graphical user interface, an EODatabaseDataSource generally forces a connection to an EOCooperatingObjectStore when it's unarchived from a nib or component. Thus, the EOObjectStoreCoordinator may never post a CooperatingObjectStoreNeeded notification.

Inside EODatabaseContext

An EODatabaseContext performs a database operation using an EODatabaseChannel. When a database context receives a message that requires database interaction (such as **objectsWithFetchSpecification**), it attempts to obtain a channel to perform the corresponding database operation as follows:

1. If the database context has a registered channel that isn't busy (that is, a channel that doesn't have a fetch in progress), it uses the available channel.

2. If the EODatabaseContext doesn't have an available channel (either because all the channels are busy or because the context doesn't have any channels), it posts a DatabaseChannelNeededNotification (EODatabaseChannelNeededNotification in Objective-C)

so that another object can register a new channel. After posting the notification, the context checks its list of registered channels a second time. If it finds an available channel (because an object registered a new channel), it uses the newly registered channel.

3. If the database context doesn't have any registered channels after posting a DatabaseChannelNeededNotification, it creates one, puts it in its list of registered channels, and uses the new channel to perform the database operation.

Note: By default, an EODatabaseContext has one EODatabaseChannel, but you can register additional channels programmatically. For more information, see the chapter "Connecting to a Database" on page 173.

Substituting a Custom EOCooperatingObjectStore

There are two approaches to providing a custom EOCooperatingObjectStore to an EOObjectStoreCoordinator:

- Tell EODatabaseContext what class to register.

  If the EOCooperatingObjectStore is a subclass of EODatabaseContext, you can simply tell EODatabaseContext to register instances of the subclass instead of EODatabaseContext instances. Use the EODatabaseContext static method **setContextClassToRegister** (**setContextClassToRegister:** class method in Objective-C) to specify your subclass.

- Register the custom EOCooperatingObjectStore yourself.

  To register your own cooperating store, add yourself as an observer of CooperatingObjectStoreNeeded notifications. When you receive a notification, create an instance of your custom store and use the EOObjectStoreCoordinator method **addCooperatingObjectStore** (**addCooperatingObjectStore:** in Objective-C) to register your cooperating store with the coordinator. To prevent EODatabaseContext from registering competing object stores, invoke the EODatabaseContext static method **setContextClassToRegister** with **null** (**nil** in Objective-C) as the argument.

For more information, see the EODatabaseContext class specification in the *Enterprise Objects Framework Reference*.

Editing Context Configurations

Recall that by default, each nib has its own EOEditingContext and that nibs share an EOObjectStoreCoordinator (see "Sharing Editing Contexts and Coordinators" on page 150). In this default configuration, each *peer* editing context has its own object graph. So for example, a single database row can be represented by separate enterprise object instances in different editing contexts. Changes to an object in one editing context don't affect the corresponding object in another editing context until all changes are successfully saved through their shared EOObjectStoreCoordinator. At that time the objects in all editing contexts are synchronized with the committed changes.

This arrangement is useful when an application allows the user to edit multiple independent *documents*. For example, imagine an Application Kit application that creates and modifies video rental records. Each rental is represented by a window that is loaded from the same nib.

You can implement variations on the default configuration to:

- Use one EOEditingContext for multiple nibs.

  In this scenario, multiple nibs have the same object graph and therefore see each other's changes to objects immediately.

- Use nested EOEditingContexts.

  This configuration is useful in a "drill down" user interface where, for example, changes in a nested dialog box can be okayed or canceled.

Using One Editing Context for Multiple Nibs

Ordinarily, changes to objects in one EOEditingContext aren't immediately reflected in the objects of another editing context. Figure 40 shows how this plays out in an application.

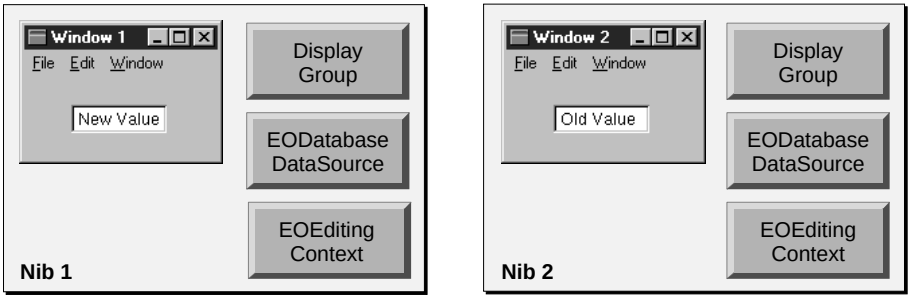


Figure 40. Different Nibs Use Different EOEditingContexts

In an application where changes in Window 1 should be immediately reflected in Window 2, both nibs should use the same editing context.

To use one editing context for multiple nibs, use the EOEditingContext static method **setSubstitutionEditingContext** (the **setSubstitutionEditingContext:** class method in Objective-C). You use this method to substitute the specified editing context for the one associated with a nib file you're about to load. This method causes all of the connections in your nib file to be redirected to the specified editing context.

For example, if Nib 1 in the figure above is loaded before Nib 2, you could invoke the following code before loading Nib 2 to set its EOEditingContext to the same one in Nib 1.

In Java:

```
EODisplayGroup displayGroup;
  // Assume that displayGroup is the display group
  // from Nib 1 and that Nib 1 has already been loaded.
EOEditingContext editingContext;

editingContext =
displayGroup.dataSource().editingContext();
EOEditingContext.setSubstitutionEditingContext(editingCo
ntext);
NSApplication.loadNibNamed("Nib2", this);

// Restore the default behavior
EOEditingContext.setSubstitutionEditingContext(null);
```

In Objective-C:

```
EODisplayGroup *displayGroup;
  // Assume that displayGroup is the display group
  // from Nib 1 and that Nib 1 has already been loaded.
EOEditingContext *editingContext;

editingContext = [[displayGroup dataSource]
editingContext];
[EOEditingContext
setSubstitutionEditingContext:editingContext];
[NSApplication loadNibNamed:@"Nib2" owner:self];

// Restore the default behavior
[EOEditingContext setSubstitutionEditingContext:nil];
```

After loading a nib with a substitution editing context, you should restore the default behavior by setting the substitution editing context to **null** (**nil** in Objective-C). Then when nibs are loaded in the future, their editing contexts are simply unarchived and aren't replaced.

Using Nested Editing Contexts

EOEditingContexts can be nested, allowing a user to make edits to an object graph in one editing context and then discard or commit those changes to another object graph (which, in turn, may commit them to an external store). For example, Figure 41 shows a drill-down user interface that can be implemented with nested editing contexts. Before users save rental information, they can supply optional comments about the rental. Canceling the Comments panel reverts the rental to its pre-comments state. Okaying the Comments panel incorporates the comments into the rental.

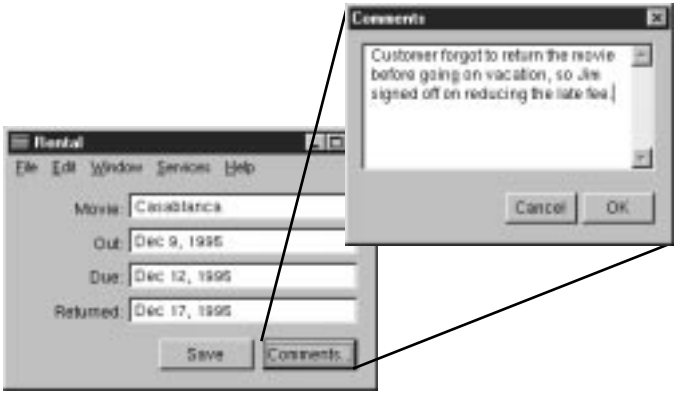


Figure 41. A Drill-Down User Interface

To implement the drill-down behavior of this interface, the editing context for the Comments window sits on top of the Rental window's editing context as shown in Figure 42. In this configuration, the Rental window's editing context is the parent object store of the Comment's editing context.

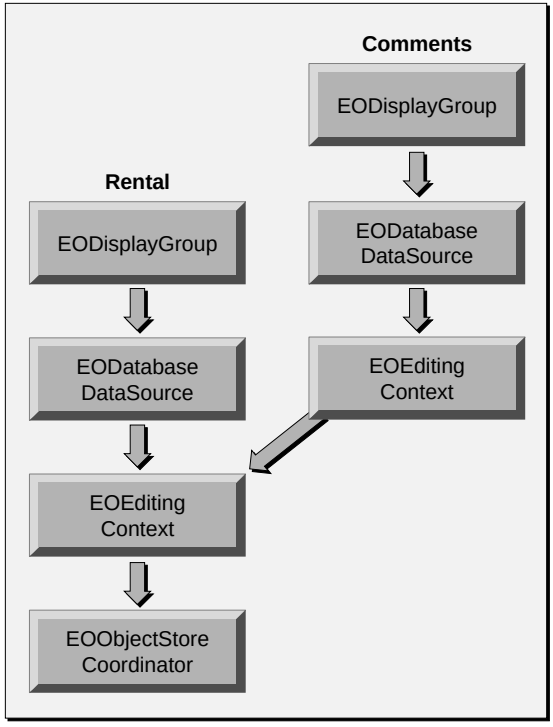


Figure 42. Nested Editing Context Configuration

To set up a nested editing context configuration, use the EOEditingContext static method **setDefaultParentObjectStore** (**setDefaultParentObjectStore:** in Objective-C). For example, Figure 43 shows the nibs for the Rental and Comments windows.

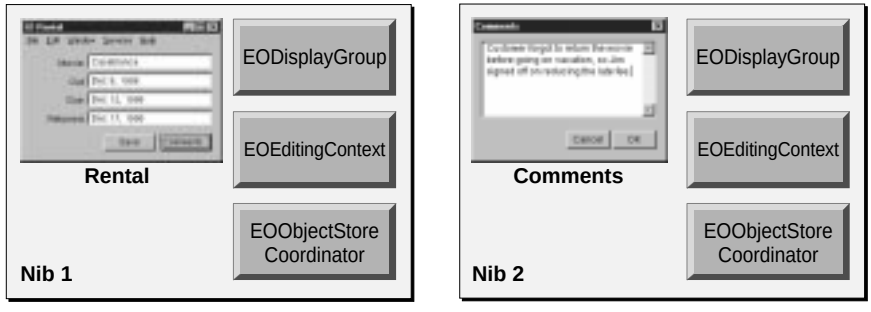


Figure 43. Nibs for the Rental and Comments Windows

Before loading Nib 2, the Rental application should invoke the following code to assign the Rental editing context as the parent object store of the Comments editing context.

In Java:

```
EODisplayGroup rentalsDisplayGroup;
  // Assume that rentalsDisplayGroup is the display group
  // from Nib 1 and that Nib 1 has already been loaded.
EOEditingContext editingContext;

editingContext =
displayGroup.dataSource().editingContext();
EOEditingContext.setDefaultParentObjectStore(editingCont
ext);
NSApplication.loadNibNamed("Nib2", this);

// Restore the default behavior
EOEditingContext.setDefaultParentObjectStore(null);
```

In Objective-C:

```
EODisplayGroup *rentalsDisplayGroup;
  // Assume that rentalsDisplayGroup is the display group
  // from Nib 1 and that Nib 1 has already been loaded.
EOEditingContext *editingContext;

editingContext = [[displayGroup dataSource]
editingContext];
[EOEditingContext
setDefaultParentObjectStore:editingContext];
[NSApplication loadNibNamed:@"Nib2" owner:self];

// Restore the default behavior
[EOEditingContext setDefaultParentObjectStore:nil];
```

After loading a nib with an editing context substituted as the default parent object store, you should restore the default behavior by setting the default parent EOObjectStore to **null** (**nil** in Objective-C). Then when nibs are loaded in the future, their editing contexts are simply connected to the default EOObjectStoreCoordinator.

Object Store Coordinator Configurations

Recall that by default, all the EOEditingContexts in an application share the same EOObjectStoreCoordinator (see “Sharing Editing Contexts and Coordinators” on page 150). In this default configuration, all of a coordinator’s editing contexts are synchronized with one another after any of the editing contexts save changes. Also, the editing contexts share underlying database connections wherever possible. This default behavior is typically what you want, but there are a some rare situations in which you might need more than one EOObjectStoreCoordinator.

All entity names must be unique within the scope of an EOObjectStoreCoordinator, so you need multiple coordinators when your application uses more than one connection to a database and each connection uses entities with the same name. For example, the following scenarios require multiple coordinators:

- An application that performs two types of tasks—regular user tasks and administrative tasks

  The different types of tasks require different connections to the database. Regular user tasks go through a database connection that uses a regular user login while administrative tasks go through a database connection that uses a special administrative login. The two connections use different connection dictionaries, but otherwise use the same models. Consequently, the each connection uses the same entities.

- A WebObjects application that requires users to log in with their own login information

  In this scenario, you’d set up a database connection for each user session. Here, too, the database connections use different connection dictionaries, but otherwise use the same models.

- An application that requires multiple, simultaneous transactions open on the same database

  Because the transactions use the same model (and potentially the same connection information), they require their own connections to the database.

As shown in Figure 44, using an additional coordinator influences the number of database connections your application maintains.

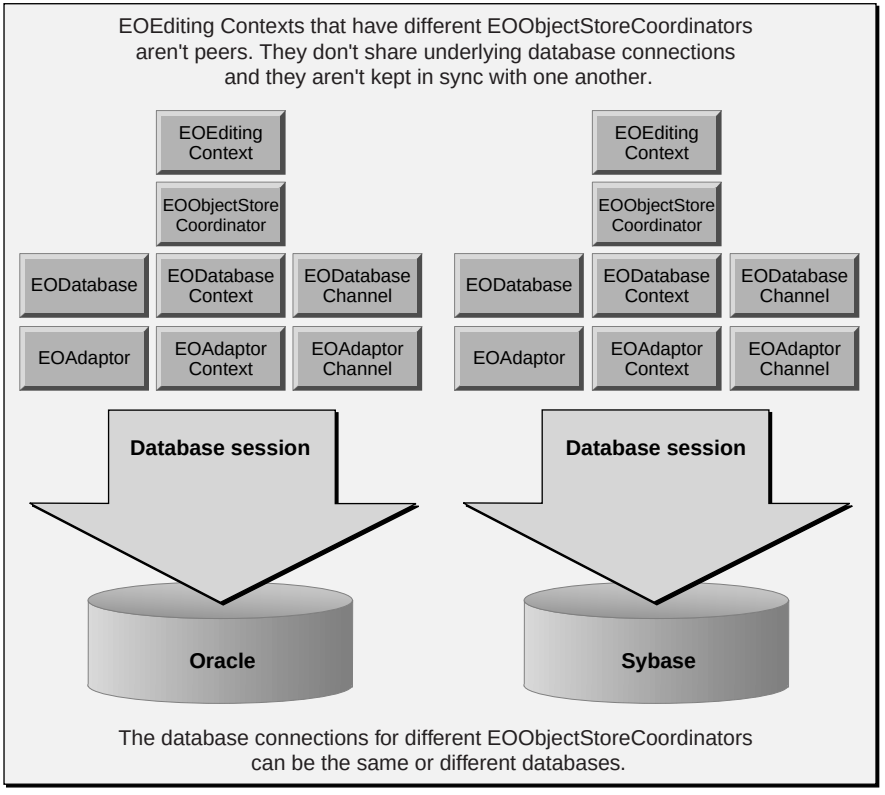


Figure 44. Multiple EOObjectStoreCoordinators

The following sections describe how to create multiple coordinators. After you create an EOObjectStoreCoordinator, it takes care of setting up its underlying network of objects as described in the sections "Inside EOObjectStoreCoordinator" on page 156 and "Inside EODatabaseContext" on page 157.

Setting Up Multiple Coordinators Programmatically

If you are creating your EOEditingContexts programmatically, assigning unique EOObjectStoreCoordinators wherever necessary is straightforward. You simply:

In Java:

```
EOObjectStoreCoordinator coordinator =
  new EOObjectStoreCoordinator();
EOEditingContext ec = new EOEditingContext(coordinator);
```

In Objective-C:

```
EOObjectStoreCoordinator *coordinator =
  [[[EOObjectStoreCoordinator alloc] init] autorelease];
EOEditingContext *ec = [[EOEditingContext alloc]
  initWithParentObjectStore:coordinator];
```

Setting Up Multiple Coordinators Using Nibs

If you are unarchiving your EOEditingContexts from nib files, you can specify a unique EOObjectStoreCoordinator using the EOEditingContext method **setDefaultParentObjectStore** (**setDefaultParentObjectStore:** in Objective-C) as follows:

In Java:

```
EOObjectStoreCoordinator coordinator =
  new EOObjectStoreCoordinator();
EOEditingContext.setDefaultParentObjectStore(
  coordinator);
NSApplication.loadNibNamed("MyNib", this);
EOEditingContext.setDefaultParentObjectStore(null);
```

In Objective-C:

```
EOObjectStoreCoordinator *coordinator =
  [[[EOObjectStoreCoordinator alloc] init] autorelease];
[EOEditingContext
  setDefaultParentObjectStore:coordinator];
[NSApplication loadNibNamed:@"MyNib" owner:self];
[EOEditingContext setDefaultParentObjectStore:nil];
```

After setting the default object store coordinator, new editing contexts (such as the one being unarchived from the nib) use the new EOObjectStoreCoordinator. After loading the nib, set the default parent object store back to the default EOObjectStoreCoordinator by sending a **setDefaultParentObjectsStore** message with **null** (**nil**) as the argument.

Accessing Multiple Databases

Enterprise Objects Framework applications access multiple databases almost transparently. Simply make different models for each database, and then you can create relationships from an entity in one database to an entity in another. In your application, you can fetch enterprise objects from different databases into the same object graph without any extra work. See the chapter "Using EOModeler" for more information.

However, there are a couple of pitfalls that can occur when you're working with more than one database:

- Enterprise Objects Framework doesn't implement a two-phase commit.
- EODatabaseDataSources with models for different databases can erroneously rendezvous on the same EODatabaseContext.

The following sections describe these problems and what you can do about them.

Getting By Without Two-Phase Commit

When an EOEditingContext has changes that need to be saved in multiple databases, the editing context's underlying EOObjectStoreCoordinator guides its EOCooperatingObjectStores (usually EODatabaseContexts) through a multi-pass save protocol in which each cooperating store saves its own changes and forwards remaining changes to other cooperating stores.

Although a coordinator manages objects from multiple repositories, it doesn't guarantee consistent updates when saving changes across object stores. If your application requires guaranteed distributed transactions, you can either provide your own solution by creating a subclass of EOObjectStoreCoordinator that integrates with a TP monitor, use a database server with built-in distributed transaction support, or design your application to write to only one object store per save operation (though it may read from multiple object stores). For more discussion of this subject, see the EOObjectStoreCoordinator class specification in the *Enterprise Objects Framework Reference*.

Preventing Database Context Rendezvousing

As described in "Database Context Rendezvousing" on page 151, EODatabaseDataSources automatically rendezvous on the same EODatabaseContexts to minimize the number of database connections an application creates. For example, in an application that accesses only one database, all the database data sources share the same database context by default. In an application that accesses two databases, there are two database contexts by default (one for each database); and as shown in Figure 45, a database data source uses one or the other of the database contexts depending on the database with which its entity is associated.

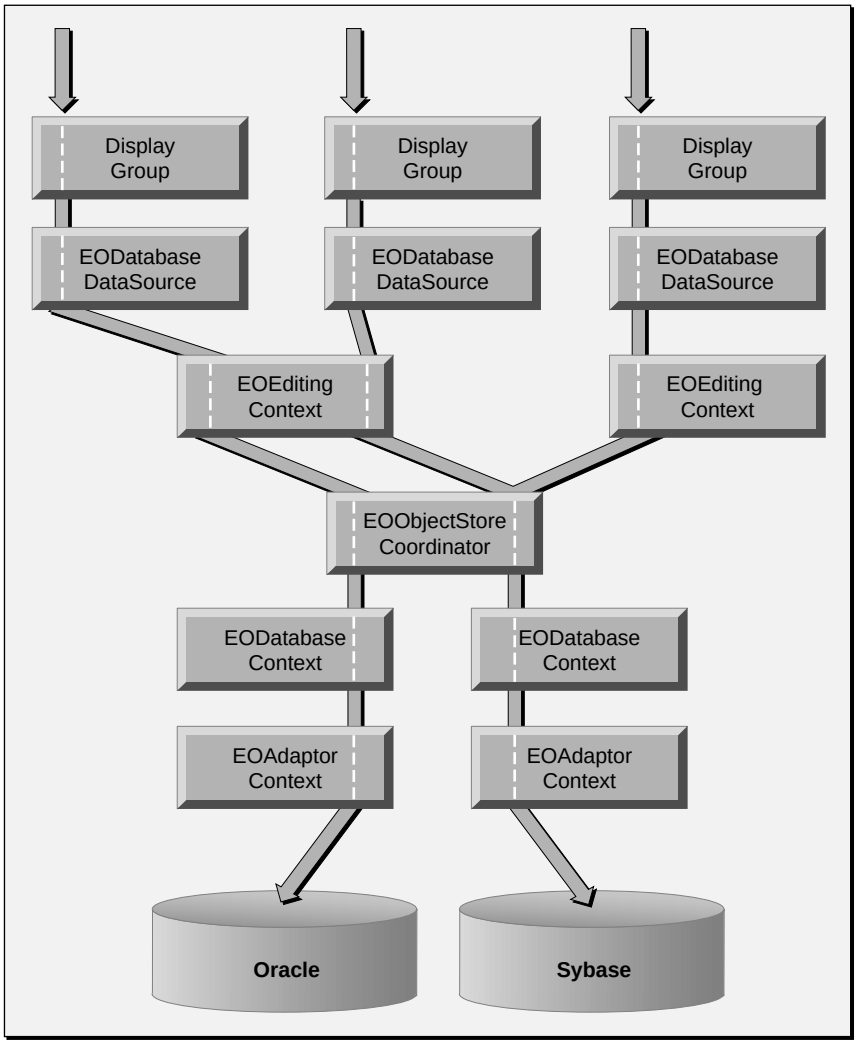


Figure 45. Sharing EODatabaseContexts

Enterprise Objects Framework determines whether or not an database data source should rendezvous with an existing database context based on the data source's model. If the data source's model is *compatible* with an database context's model, then the database data source can use the database context.

Two models are compatible when their connection dictionaries are equal. Thus, if you don't assign connection information to your models, database data sources can erroneously rendezvous with the same database context. For example, suppose an application uses two databases. The application contains two EODisplayGroups, each representing an entity from a different database. Further suppose that the models for the databases don't contain any connection information because the application requires the user to supply valid login information. In this scenario, the database data sources for each display group rendezvous on the same database context, which causes an error. Here's how it happens:

1. The first database data source is unarchived. During unarchiving, it connects a database context using the method **registeredDatabaseContextForModel** (**registeredDatabaseContextForModel:editingContext:** in Objective-C). This EODatabaseContext static method (class method in Objective-C) checks to see if a database context that can service the data source's model already exists. Since this is the first database data source to be unarchived, there isn't an database context available at all, so one is created.

2. In the process of creating the database context, the adaptor specified in the model is loaded. An EOAdaptor is instantiated, and the new database context is associated with this adaptor.

3. The second database data source is unarchived. When **registeredDatabaseContextForModel** is invoked this time, it returns the existing database context because the models for the two data sources are considered compatible.

At this point, two database data sources share a database context that represents a single database session; but two database contexts are actually needed. When the second data source attempts to interact with its database, it fails because it's using a connection to the wrong database.

The simplest way to prevent rendezvousing is to assign distinct connection dictionaries to your model files. You don't have to assign complete connection dictionaries. The dictionaries associated with different databases simply must differ in one or more entries.

Alternatively, you can programmatically assign complete connection dictionaries to your models before any EODatabaseDataSource objects are created—at application initialization time, for example. The best way to access your models is through the default EOModelGroup. The **models** method returns an array of all the EOModels used by the application.

Chapter 6

Connecting to a Database

Normally, you don't have to worry about making connections to the database, because Enterprise Objects Framework connects to the database automatically for you. However, there are times when you may need to intervene. This chapter describes how Enterprise Objects Framework manages database connections and how you can customize the process. It's organized into the following sections:

- "When Database Connections Are Opened and Closed" (page 175) describes when applications open and close database connections.
- "Logging into a Database" (page 176) describes the process of getting and validating database connection information. It answers the questions "How do I set connection information that's not in a model?" and "How can I suppress an adaptor's login panel in an OpenStep application?"
- "Limiting the Number of Database Connections" (page 180) describes how to close database connections that aren't in use.
- "Using Multiple EODatabaseChannels" (page 184) describes how to avoid "busy channel" fetching conflicts.
- "Character Encodings" (page 185) describes how to tell both the database and the adaptor what encoding to use.

When Database Connections Are Opened and Closed

If you're using an EOEditingContext in your application, Enterprise Objects Framework connects to the database automatically. It makes a connection the first time a database operation is initiated, and reuses that same connection for subsequent database operations.

Enterprise Objects Framework doesn't close its connections to the database. It leaves the connections open until the application terminates. If you need to close database connections that aren't in use, see the section "Limiting the Number of Database Connections" on page 180.

Logging into a Database

An EOAdaptor defines how an application logs into a database through a dictionary of connection information. The keys of this dictionary identify the login information the server expects; the values associated with those keys are the values that the adaptor tries when logging in. The number of connection dictionary keys and the keys themselves vary from adaptor to adaptor. For example, the dictionary keys required by the Sybase adaptor are **databaseName**, **userName**, **password**, and **hostName**. For more information on connection dictionary keys, see the class specification for the adaptor you're using.

An EOAdaptor object must have a valid connection dictionary before its application can connect to a database. There are three ways you can provide one:

- Store the connection information in a model file.
- Run the adaptor's login panel so that the user can enter login information.
- Set the connection dictionary programmatically.

The approach you choose to provide connection information hinges on the following questions:

- Do all users log in with the same connection information?
- Is any connection information sensitive?
- Does the application have an NSApplication object?
- Are users database savvy?
- Do I have time to write a custom login mechanism?

The following sections describe how to choose between and implement each of these approaches.

Storing the Connection Information in a Model File

This approach is useful when all users log in with the same connection information. It doesn't require any code, and users aren't exposed to the login process. However, you shouldn't use this approach to store connection information that is sensitive. Models are stored in an ASCII file format, and connection information is stored unencrypted. However, you can store some connection information in a model file, and use another approach to get the rest (such as requiring the user to enter a database password.)

To implement this approach just use EOModeler to add connection information to a model (see the book *Enterprise Objects Framework Tools and Techniques*).

All EOAdaptor objects that are created behind the scenes (that is, all adaptors that Enterprise Objects Framework creates automatically) are created with **adaptorWithModel** (**adaptorWithModel:** in Objective-C). This method initializes a new adaptor's connection dictionary with the connection information in the specified model. If the model's connection information is valid, no further action is required. Only adaptors that you create programmatically with **adaptorWithName** (**adaptorWithName:** in Objective-C) don't take advantage of a model's connection information.

Storing Partial Information in a Model File

When you store connection information in a model file, you don't have to store a complete connection dictionary. For example, you could store everything but the user name and password. An EOAdaptor is still initialized with the model's connection information. You can supply the missing information at run-time, either programmatically or by running the adaptor's login panel.

Running the Adaptor's Login Panel

Adaptors provide login panels for use by Enterprise Objects Framework applications. Running the login panel is a simple, no-code approach that you can use in Application Kit applications. It's useful when users have different connection information—different user names and passwords, for example. It can't, however, be used in applications that don't use the Application Kit—in other words, not in command-line or web applications.

One disadvantage of the adaptors' login panels is that they aren't configurable. Each adaptor exposes specific connection keys in the login panel. For example, the Oracle login panel has fields for server ID, user name, and password. If the panel contains fields (such as server ID) that you don't want users to see, you should use another approach. Similarly, a login panel may not provide an interface for all of an adaptor's connection keys. For example, the Oracle adaptor defines keys for language and database encoding that don't exist in the Oracle login panel.

Enterprise Objects Framework automatically runs the login panel when an EOAdaptor object doesn't have a valid connection dictionary. For example, suppose you have an EODisplayGroup that's configured to *fetch on load*. Before a user performs a single action, the application:

- Creates a network of objects under the display group
- Connects to the database
- Performs a database operation to retrieve enterprise objects

If your model doesn't have valid connection information, the application runs a login panel until the user enters valid connection information or cancels the panel. Specifically, when an EODatabaseContext is about to use an EODatabaseChannel to interact with a database:

1. The EODatabaseContext checks to see if the EODatabaseChannel's underlying EOAdaptorChannel is open.

2. If the EOAdaptorChannel isn't open, the EODatabaseContext attempts to open the adaptor channel by sending it an **openChannel** message.

3. If **openChannel** fails, the EODatabaseContext runs the adaptor's login panel by sending the EOAdaptorChannel's adaptor a **runLoginPanelAndValidateConnectionDictionary** message.

Canceling the panel has the effect of canceling whatever operation is in progress. In the example above, canceling the panel cancels the fetch, and the user interface opens without any data to display.

Suppressing the Login Panel

You can suppress the database adaptor's login panel by implementing the EODatabaseContext delegate method

databaseContextWillRunLoginPanelToOpenDatabaseChannel (**databaseContext:willRunLoginPanelToOpenDatabaseChannel:** in Objective-C). For more information, see the EODatabaseContext class specification in the *Enterprise Objects Framework Reference*.

Alternatively, you can prevent an application from even attempting to run a login panel by ensuring that its EOAdaptor objects have valid connection dictionaries. Before an application's first attempt to connect to a database, send an **assertConnectionDictionaryIsValid** message to an EOAdaptor. If the adaptor doesn't have sufficient information to log in (for example, it's common to leave the user name and password unspecified in the model file), **assertConnectionDictionaryIsValid** throws an exception. In your exception handler, set the adaptor's connection dictionary programmatically.

Setting the Connection Dictionary Programmatically

If you don't store connection information in your model and your application doesn't have an Application Kit user interface, you have to set an adaptor's connection dictionary programmatically. It requires code, but this approach is also useful if you want to implement a login panel that's tailored to your application. For example, you can implement a login component for a WebObjects application, a custom login panel for an Application Kit Framework application, or a mechanism for tools and background processes that gets connection information from command line arguments.

How you get the connection information is up to you. Once you get it, setting the connection dictionary is simple:

1. Insert the connection information into an NSDictionary object using adaptor-defined keys.

2. Assign the dictionary to the adaptor using the EOAdaptor method **setConnectionDictionary** (**setConnectionDictionary:** in Objective-C).

3. Verify that the connection dictionary is valid using the EOAdaptor method **assertConnectionDictionaryIsValid**.

For more information, see the EOAdaptor class specification in the *Enterprise Objects Framework Reference*.

Getting Partial Information from a Model File

If you store some connection information in a model file, an EOAdaptor is object is initialized with an incomplete connection dictionary.

To supply the missing information:

1. Get the adaptor's incomplete connection dictionary using the EOAdaptor method **connectionDictionary**.
2. Create a mutable copy.
3. Insert the missing key-value entries.
4. Assign the new, complete connection dictionary to the EOAdaptor with **setConnectionDictionary** (**setConnectionDictionary:** in Objective-C).

Limiting the Number of Database Connections

By default, an Enterprise Objects Framework application uses one connection to the database—that is, all of an application's display groups, EODatabaseDataSources, EOEditingContexts, and EODatabaseDataSources share the same database connections. However, if an application accesses multiple databases (an Oracle database and a Sybase database, for example), Enterprise Objects Framework establishes one database connection for each database, and these connections are shared as shown in Figure 46.

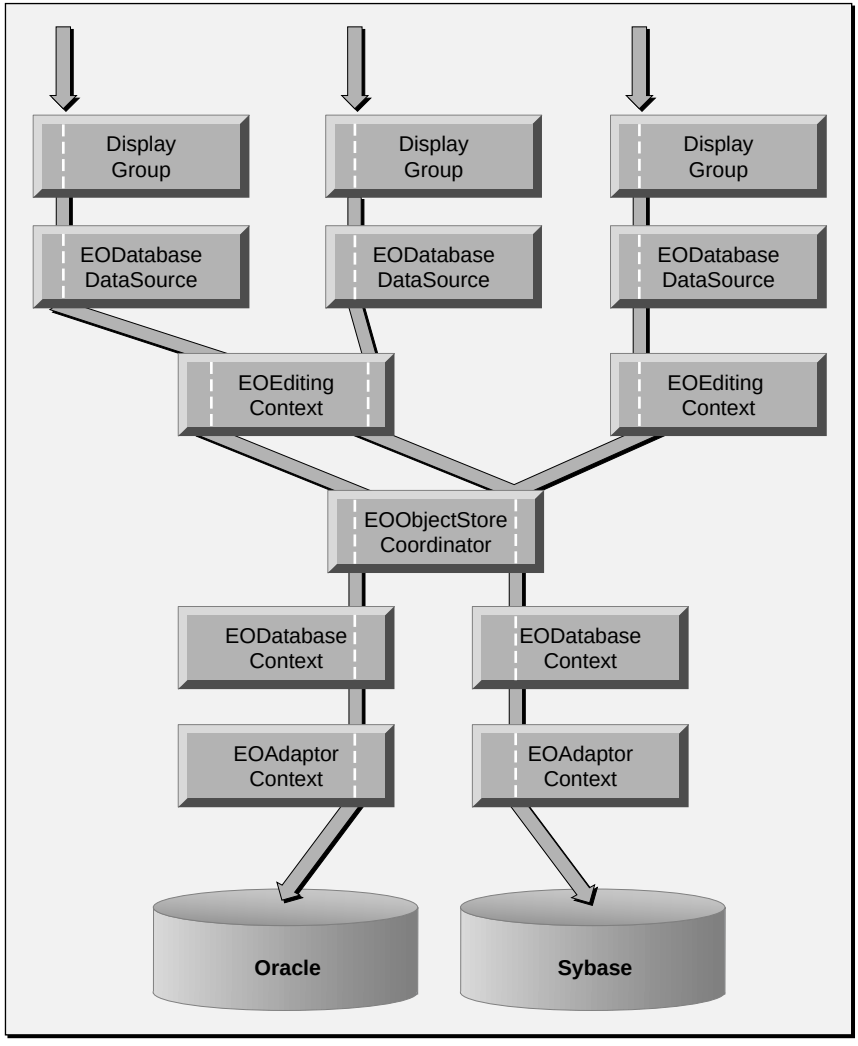


Figure 46. Sharing Database Connections

Because Enterprise Objects Framework uses the minimum number of connections by default, you don't need to do anything to limit the number of connections an application uses. However, you can close connections when they aren't in use.

Closing Database Connections

Enterprise Objects Framework doesn't close database connections. If many copies of an application are likely to be running at the same time, you may run out of database connections. You can reduce the likelihood of running out if you close connections when they aren't in use. A good time to close an EODatabaseChannel is after a specified period of inactivity. The following method demonstrates the process:

In Java:

```
public void closeChannels() {
  int i, contextCount, j, channelCount;
  NSArray contexts;
  EOObjectStoreCoordinator coordinator;

  coordinator =
      (EOObjectStoreCoordinator)EOObjectStoreCoordinator.
          defaultCoordinator();

  contexts = coordinator.cooperatingObjectStores();
  contextCount = contexts.count();
  for (i = 0; i < contextCount; i++) {
      NSArray channels =
          ((EODatabaseContext)contexts.objectAtIndex(i)).
              registeredChannels();
      channelCount = channels.count();
      for (j = 0; j < channelCount; j++) {
          ((EODatabaseChannel)channels.objectAtIndex(j)).
              adaptorChannel().closeChannel();
      }
  }
}
```

In Objective-C:

```
- (void)closeChannels
{
  int i, contextCount, j, channelCount;
  NSArray *contexts;
  EOObjectStoreCoordinator *coordinator;

  coordinator = [EOObjectStoreCoordinator
defaultCoordinator];

  contexts = [coordinator cooperatingObjectStores];
  contextCount = [contexts count];
  for (i = 0; i < contextCount; i++) {
      NSArray *channels = [(EODatabaseContext *)
      [contexts objectAtIndex:i] registeredChannels];
      channelCount = [channels count];

      for (j = 0; j < channelCount; j++) {
          [[(EODatabaseChannel *)
              [channels objectAtIndex:j] adaptorChannel]
              closeChannel];
      }
  }
}
```

The **closeChannels** method gets the EODatabaseContexts from the default EOObjectStoreCoordinator. Then it gets the EODatabaseChannels registered with each EODatabaseContext. To close the database connection managed by an EODatabaseChannel, **closeChannels** sends the channel's EOAdaptorChannel a **closeChannel** message.

The next time a channel is needed, its EODatabaseContext reopens it automatically.

The **closeChannels** method above assumes that the application only has one EOObjectStoreCoordinator. If your application has multiple coordinators, you would repeat the process for each coordinator. It also assumes that all of the EOCooperatingObjectStores managed by the coordinator are EODatabaseContexts, which is nearly always the case. An EOObjectStoreCoordinator uses only EODatabaseContexts unless you substitute your own EOCooperatingObjectStore subclass. (For more information about EOCooperatingObjectStores, see the chapter "Application Configurations" on page 141.)

Using Multiple EODatabaseChannels

By default, an EODatabaseContext uses one EODatabaseChannel. However, occasionally your application needs more channels. Conflicts due to "busy channels" can occur when an EODatabaseContext needs to perform a database operation and its EODatabaseChannel is already fetching. Most such conflicts are manifestations of inefficient database access and can be avoided. For more information, see the section "Cautions in Implementing Accessor Methods" on page 102 in the chapter "Designing Enterprise Objects." However, if you can't eliminate fetching conflicts, using additional EODatabaseChannels is an option.

When an EODatabaseContext needs a new channel because all its current channels are busy, it posts an EODatabaseChannelNeededNotification. If you add yourself as an observer of this notification, you can create new EODatabaseChannels on demand. (For more information on registering for notifications, see the NSNotification and NSNotificationCenter class specifications in the *Foundation Reference*.)

Note: You should set an upper limit on the number of EODatabaseChannels your application registers with an EODatabaseContext. It's very unusual for an EODatabaseContext to require more than two or three EODatabaseChannels.

The following code examples demonstrate creating a new EODatabaseChannel and registering it with an EODatabaseContext:

In Java:

```
EODatabaseContext context; // Assume this exists
EODatabaseChannel channel = new
EODatabaseChannel(context);
if (channel) context.registerChannel(channel);
```

In Objective-C:

```
EODatabaseContext *context; // Assume this exists
EODatabaseChannel *channel = [[EODatabaseChannel alloc]
      initWithDatabaseContext:context];
if (channel) [context registerChannel:channel];
```

The EODatabaseChannel constructor can return **null** if no more channels can be associated with the EODatabaseContext. Similarly, in Objective-C, the EODatabaseChannel method **initWithDatabaseContext:** can return **nil** if no more channels can be associated with the EODatabaseContext. Some database servers and their corresponding adaptors don't support multiple channels per context. For example, the Sybase adaptor only supports one EODatabaseChannel per EODatabaseContext.

Character Encodings

An Enterprise Objects Framework adaptor and a database must communicate with one another using the same character encoding. For example, if the database sends data to your application using the EUC (Japanese) encoding, your application must interpret the data as EUC-encoded. Consequently, you have to tell both the database and the database adaptor what encoding to use.

Choosing an Encoding

In choosing the encoding to use, you should attempt to minimize the amount of data conversion the database has to perform. For example, if the database stores EUC-encoded data, you should configure the database and adaptor to communicate with one another using the EUC encoding so the database doesn't have to perform a conversion. On the other hand, if your database stores data in an encoding that Enterprise Objects Framework doesn't support (such as EBCDIC), the database must convert its data to a supported encoding before sending it to your application. Similarly, it must convert data that it receives from your application into the encoding it uses for data storage. See the "Types and Constants" section of the *Foundation Reference* for a complete list of supported encodings.

You should also attempt to minimize information loss by choosing an encoding that is as rich as the encoding the database uses for data storage. For example, if you choose a 7 bit ASCII encoding for communication when the database stores data in Unicode, the Unicode-encoded character 'Á' loses its accent during conversion.

Setting an Adaptor's Character Encoding

By default, Enterprise Objects Framework adaptors send and expect to receive data that is encoded with the default C string encoding. Since this encoding is unlikely to match the encoding the database uses for storage, you usually have to set it to a different encoding.

To change an adaptor's character encoding from the default, add a **databaseEncoding** entry to the adaptor's connection dictionary specifying the encoding. The adaptors expect the **databaseEncoding** entry to contain the localized name of an encoding. To get the localized name of an encoding, use the NSStringReference static method **localizedNameOfStringEncoding** (in Objective-C, NSString's class method **localizedNameOfStringEncoding:**).

Setting the Database Character Encoding

The default encoding a database uses for communicating with applications is database-dependent. Check your database server's documentation for more information.

If you need to use an encoding other than the default, Enterprise Objects Framework adaptors define adaptor-specific connection keys for setting the encoding. For example, the Sybase adaptor has the key **LC_ALL** and the Oracle adaptor has the key **NLS_LANG**. To set the encoding the database should use to send data to and receive data from your application, add an entry to the adaptor's connection dictionary for the adaptor-specific key. Check your database server's documentation for the available character encodings.

Note: The **databaseEncoding** and the adaptor-specific encoding entries in a connection dictionary must specify the same encoding. However, the string that identifies the encoding may differ. For example, to tell a Sybase database to use the ISO Latin 1 encoding, you set the **LC_ALL** connection key to "iso-1" and the **databaseEncoding** connection key to "ISO Latin-1".

Chapter 7

Behind the Scenes

Using Application Kit applications as examples, this chapter answers the following questions about what Enterprise Objects Framework does behind the scenes:

- What is the sequence of events when objects are fetched from the database?
- How does an EOEditingContext manage changes to its objects?
- What happens when changes to objects are saved to the database?
- How does Enterprise Objects Framework manage transactions?

Enterprise Objects Framework provides hooks so that your code can intervene in each of these scenarios. In addition to describing what happens behind the scenes in an Enterprise Objects Framework application, this chapter lists the delegate methods and notifications your code can use at every stage of an application to do custom processing.

Most of the information in this chapter can also be applied to other types of applications (command-line and web applications), but some of the details vary from what's illustrated here with Application Kit applications.

Fetching Objects

This section describes the sequence of events that occurs when objects are fetched from the database. It's broken down into the following major sections:

- "EODisplayGroup Receives a fetch Message" (page 191)
- "Inside EODatabaseContext" (page 192)
- "Inside EODatabaseChannel" (page 195)

Figure 47 provides a high-level view of what happens in Enterprise Objects Framework when you fetch an object.

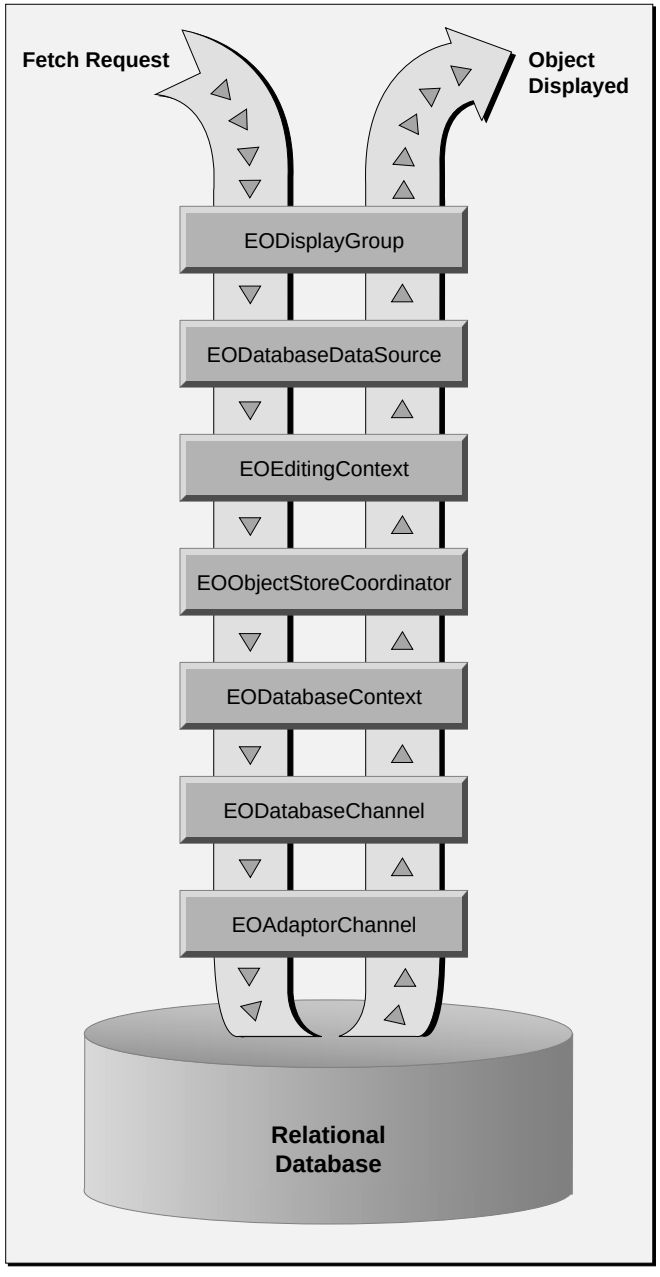


Figure 47. What Happens During a Fetch

Note: Figure 47 is intended to show the flow of data, not the Enterprise Objects Framework architecture as such. For a graphical depiction of the Enterprise Objects Framework architecture, see the chapter "What Is Enterprise Objects Framework?" on page 19

The steps illustrated in Figure 47 are described in detail in the following sections.

EODisplayGroup Receives a fetch Message

A fetch most often begins with an EODisplayGroup receiving a **fetch** message. If you've configured the display group to "fetch on load," the fetch occurs at the end of display group initialization.

When a display group receives a **fetch** message, the following sequence of events occurs:

1. The display group sends its EODataSource a **fetchObjects** message, which for the EODatabaseDataSource subclass results in a fetch against a specific EOEntity.

2. The EODatabaseDataSource constructs an EOFetchSpecification for the database data source's EOEntity (if a fetch specification doesn't already exist) and passes the specification to its EOEditingContext in an **objectsWithFetchSpecification** message (**objectsWithFetchSpecification:** in Objective-C).

   The database data source would already have a fetch specification if you assigned one programmatically or if you assigned one in Interface Builder (or WebObjects Builder in a WebObjects application). Fetch specifications are handled differently depending on how they're configured. This discussion assumes that the fetch specification is a "regular" fetch specification—one that isn't configured to fetch raw rows, to use a custom SQL statement, or to use a stored procedure.

3. The receiving editing context forwards this request, in the form of a **objectsWithFetchSpecification** message (**objectsWithFetchSpecification:editingContext:** in Objective-C), to its parent object store, eventually reaching the root object store (which is usually an EOObjectStoreCoordinator).

An EOObjectStoreCoordinator manages one or more EODatabaseContexts or other EOCooperatingObjectStores.

4. The EOObjectStoreCoordinator determines which of its EOCooperatingObjectStores should service the fetch specification and forwards the EOCooperatingObjectStore an **objectsWithFetchSpecification** message (**objectsWithFetchSpecification:editingContext:** in Objective-C) to ask it to actually retrieve data from the database.

Note: A fetch operation is always performed with an editing context, which holds all of the objects it fetches (an editing context usually performs multiple fetches over the course of an application). Objects are kept unique within an editing context, so if an enterprise object instance already exists for a fetched row, that object is simply returned as the fetched object (possibly with updated values, as described below). Any faults created for fetched objects likewise belong to the editing context and are kept unique within it.

Inside EODatabaseContext

When an EODatabaseContext receives a fetch request in the form of an **objectsWithFetchSpecification** message (**objectsWithFetchSpecification:editingContext:** in Objective-C), it fetches a number of rows from the database, transforms them into enterprise objects, and registers them as needed with the EOEditingContext that received the initial **objectsWithFetchSpecification** message.

To do this it uses an EODatabaseChannel, whose job is specifically to fetch enterprise objects. The database channel in turn uses an EOAdaptorChannel for low-level communication with the database, along with whatever model objects—EOEntities, EOAttributes, and EORelationships—are needed to perform the fetch.

Fetching objects happens in two major steps:

1. The database context uses the database channel to select the rows in the database for which objects are being fetched. To do this, it uses the EODatabaseChannel **selectObjectsWithFetchSpecification** method (**selectObjectsWithFetchSpecification:editingContext:** in Objective-C), passing in the fetch specification. Minimally, the fetch specification identifies the EOEntity for the objects, which in turn specifies the enterprise object class to instantiate for every object fetched. If you provided the fetch specification that's used, it can also contain a qualifier that restricts the objects to fetch to those that meet specified criteria and a sort ordering with which to sort the objects. The section "Inside EODatabaseChannel" on page 195" describes in detail how the fetch specification is handled.

2. The database channel fetches each enterprise object, one at a time, as the database context repeatedly sends it the message **fetchObject**. This method uses state built up in the select step to get data for the object, create an instance if necessary, and register the new object with the editing context.

Customizing Framework Behavior

At this stage in the fetching process, there are several ways you can customize the Framework's default behavior. To get fine-grained control over a database context, you can assign a delegate to it and implement any of the following methods.

EODatabaseContext Delegate Methods

Java Method	Objective-C Method	Description
databaseContext ShouldSelectObjects	databaseContext: shouldSelectObjects WithFetchSpecification: databaseChannel:	This method is invoked right before a SELECT occurs. You can return **false** (NO) to tell the channel to skip the SELECT and return; you might want to do this to issue your own custom SQL against the adaptor.

EODatabaseContext Delegate Methods (Continued)

Java Method	Objective-C Method	Description
databaseContext ShouldUsePessimisticLock	databaseContext: shouldUsePessimisticLock WithFetchSpecification: databaseChannel:	You can use this delegate method to selectively turn off the locking of rows when you're using a pessimistic locking strategy.
databaseContext DidSelectObjects	databaseContext: didSelectObjects WithFetchSpecification: databaseChannel:	This method is invoked immediately after a SELECT occurs. You can use it to log diagnostic information or set up internal state for the coming fetch.
databaseContext ShouldFetchObjects	databaseContext: shouldFetchObjects WithFetchSpecification: editingContext:	You can use this method to satisfy the EOEditingContext's fetch request from a local cache.
databaseContext DidFetchObjects	databaseContext: didFetchObjects: fetchSpecification: editingContext:	This method is invoked after an EODatabaseContext fetches objects. You can use this method to record in a local cache the results of a fetch.

With respect to locking, note that in addition to setting an overall locking strategy, you can take advantage of EODatabaseContext's "on demand" locking feature to lock individual rows. For more information, see "Locking and Update Strategies" on page 217.

An EODatabaseContext also posts notifications that your objects can receive and react to.

EODatabaseContext Notifications

Notification	Description
DatabaseChannelNeededNotification (EODatabaseChannelNeededNotification in Objective-C)	This notification is broadcast whenever an EODatabaseContext is asked to perform an object store operation and it doesn't have an available EODatabaseChannel. Subscribers can create a new channel and add it to the EODatabaseContext at this time.

Inside EODatabaseChannel

In **fetchObject**, an EODatabaseChannel performs several tasks:

1. Before anything else can be done, the EODatabaseChannel must read some data from the database. It does so by having its EOAdaptorChannel retrieve a record for the EOEntity being fetched, including the primary key, class properties, attributes used for locking, and any foreign keys used by EORelationships.

2. The first thing the database channel does with the fetched record is to get an EOGlobalID for it from the EOEntity by invoking **globalIDForRow** (**globalIDForRow:** in Objective-C).

3. The EODatabaseChannel records a snapshot for the fetched row. This step can be fairly complicated, as there may already be a snapshot recorded under the globalID. If there isn't, the EODatabase object is simply sent a **recordSnapshotForGlobalID** message (**recordSnapshot:forGlobalID:** in Objective-C). If there is a snapshot, however, a decision must be made on how to update the recorded snapshot. You can use the EODatabaseContext delegate method **databaseContextShouldUpdateCurrentSnapshot** to intervene at this point (in Objective-C, **databaseContext: shouldUpdateCurrentSnapshot:newSnapshot:globalID: databaseChannel:**).

   If the fetch specification is set to refresh refetched objects, an ObjectsChangedInStoreNotification (EOObjectsChangedInStoreNotification in Objective-C) is posted to invalidate (refault) any existing instances corresponding to this globalID.

4. The database channel records whether the object was locked when it was selected.

5. The database channel then checks with the editing context, using **objectForGlobalID** (**objectForGlobalID:** in Objective-C), to see whether a copy of the object already exists in that context.

6. If the editing context already has an enterprise object for the global ID—and if it isn't a fault—then it's simply returned; otherwise it returns **null** (**nil** in Objective-C).

7. If the editing context has no object or fault for the globalID, the database channel invokes the EOEntityClassDescription method **createInstanceWithEditingContext** (**createInstanceWithEditingContext:globalID:zone:** in Objective-C).This method finds out what the object's class should be from the EOEntity and creates an object of that class.

   The **createInstanceWithEditingContext** method provides the enterprise object class's constructor with an editing context, the entity class description, and a globalID. You'll rarely need to use this information at this point, but you might choose to if, for example, you need to extract the primary key from the globalID to do some processing on it.

   Note that in Objective-C, the entity class description creates objects and initializes them with the method **initWithEditingContext:classDescription:globalID:**, if it exists (it uses **init** otherwise). Your enterprise object class can implement **initWithEditingContext:classDescription:globalID:** instead of simply **init** if you need to do anything with the object's editing context, class description, or globalID at this stage in the process.

8. The database channel invokes the editing context's **recordObject** method (**recordObject:globalID:** in Objective-C), in which the newly created object gets uniqued.

9. If the editing context has a fault for the globalID, the fault is cleared and initialization proceeds just as if an empty enterprise object had been created and registered.

10. To initialize the object, database channel sends the editing context an **initializeObject** message (**initializeObject:withGlobalID:editingContext:** in Objective-C), which is passed down the object store hierarchy. If the editing context is nested, it passes the message to its parent EOEditingContext. If the parent EOEditingContext has an object with a matching globalID, that object is used to initialize the child object. Otherwise, the message is passed down to the EODatabaseContext, which initializes the new instance from the appropriate snapshot and creates faults for its relationships. The EODatabaseContext's **initializeObject** method sets the

object's properties using the key-value coding method **takeStoredValueForKey** (**takeStoredValue:forKey:** in Objective-C).

11. The database channel sends the enterprise object an **awakeFromFetch** message (**awakeFromFetchInEditingContext:** in Objective-C). Custom enterprise object classes can override this method to perform additional initialization after an object has been created from a database row and initialized from database values.

    Your custom enterprise object classes can also implement the method **awakeFromInsertion** (**awakeFromInsertionInEditingContext:** in Objective-C), which is invoked immediately after your application creates a new object and inserts it into an EOEditingContext. This method lets you assign values to newly created enterprise objects. For more discussion of this topic, see the chapter "Designing Enterprise Objects" on page 65.

Customizing Framework Behavior

At this stage in the fetching process, you can intervene during step 5 above by assigning a delegate to the EODatabaseContext and implementing any of the following methods.

EODatabaseContext Delegate Methods

Java Method	Objective-C Method	Description
databaseContext FailedToFetchObject	databaseContext: failedToFetchObject: globalID:	This method is invoked when a to-one fault can't find its data in the database. This often occurs due to referential integrity problems in the database. You can use this method to intervene and take appropriate action (for example, by displaying an alert panel or initializing a fault object with new values).
databaseContext ShouldLockObject WithGlobalID	databaseContext: shouldLockObject WithGlobalID: snapshot:	This method is invoked from lockObjectWithGlobalID:editing Context:. You can use this method to implement your own locking procedure.

EODatabaseContext Delegate Methods (Continued)

Java Method	Objective-C Method	Description
databaseContext ShouldRaiseException ForLockFailure	databaseContext: shouldRaiseException ForLockFailure:	You can use this method to suppress an exception that occurred during an EODatabaseContext's attempt to lock an object.
databaseContext ShouldUpdate CurrentSnapshot	databaseContext: shouldUpdate CurrentSnapshot: newSnapshot: globalID: databaseChannel:	This method is invoked when an EODatabaseContext already has a snapshot for a row fetched from the database. You can use this method to compare the snapshots, possibly resolve conflicts, and instruct the EODatabaseContext to use the new snapshot instead of the existing one.

Flow of Data During a Fetch

The preceding sections describe what happens in Enterprise Objects Framework when you fetch objects from the database. This section describes the flow of data that occurs during a fetch.

Figure 48 shows how a new object gets instantiated with database data. The scenario it depicts is fetching a single, new object from the database.

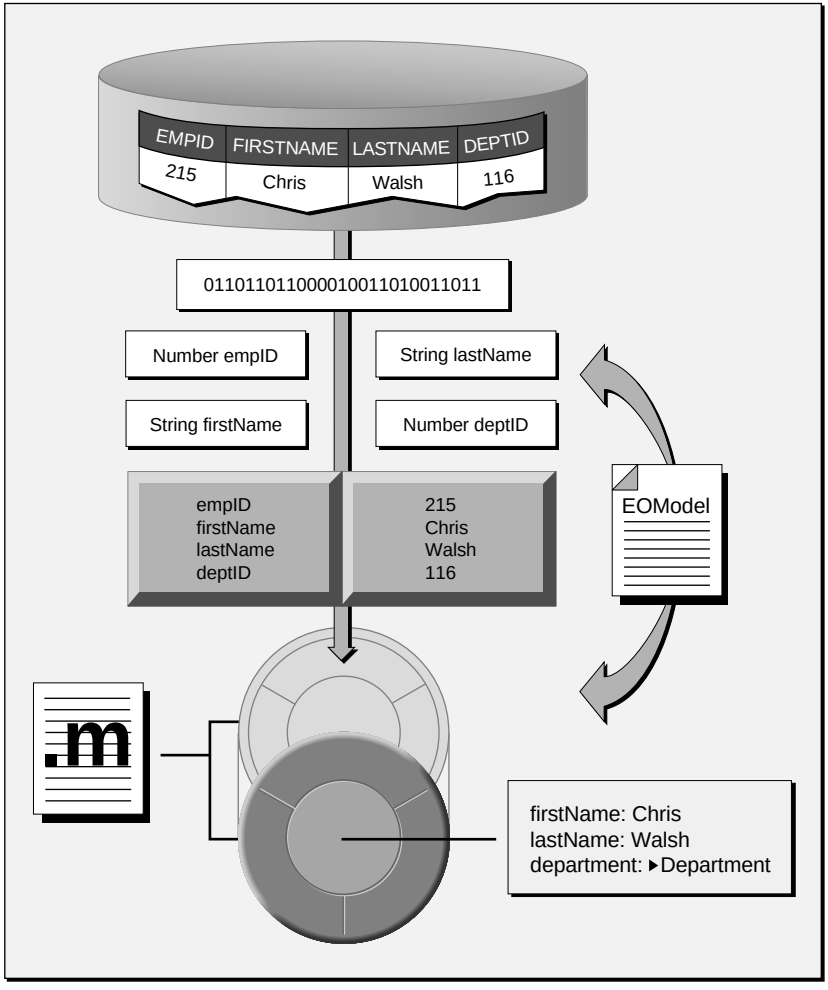


Figure 48. Flow of Data During a Fetch

This process is described in greater detail below.

The following sequence of events occurs when an object is fetched from the database:

1. A database row is fetched as raw binary data.

2. The values retrieved from the database are converted from their database-specific types to instances of standard value classes:

Java Class	Objective-C Class	Type of Data
String	NSString	character strings
NSGregorianDate	NSCalendarDate	dates and times
Number or BigDecimal (java.math)	NSNumber or NSDecimalNumber	numbers
NSData	NSData	arbitrary binary data (BLOBs)

   NULL values in the database are mapped to instances of EONullValue (EONull in Objective-C).

   Additionally, you can map external data types to custom value classes defined by your application. For more discussion of this subject, see the chapter "Advanced Enterprise Object Modeling" on page 105.

3. Once the data has been converted to objects, these objects are put in an NSDictionary. The elements of the dictionary correspond to columns (attributes) in the database table: Their names are the names of the attributes as used by the client application, and their values are the values in the database. The EOModel is used to determine the mapping from external (database) data types to internal (Objective-C) types.

   The dictionary provides a snapshot of the database row, and it's later used to initialize the enterprise object. The snapshot also comes into play when changes to the object are saved to the database; for more discussion of this topic see the section "Snapshots" on page 204.

The EOModel, which is used to convert database data to objects, is also used when a newly allocated enterprise object is initialized. Whereas the dictionary contains an entry for each of the row's columns (those returned by sending **attributesToFetch** to an EOEntity), the enterprise object initialized from the dictionary only contains the attributes that are defined in the EOModel as class properties.

4. A new enterprise object is allocated by EOEntityClassDescription as an object of the Employee class, as determined from the EOModel.

5. The enterprise object is initialized from a row snapshot, using the EOModel. Only objects that are class properties are included.

   When an enterprise object is initialized, EONullValue objects (or EONull objects in Objective-C) are passed to the object as **null** (**nil** in Objective-C) so you don't have to write code to handle NULLs.

   Also, relationship references are initialized for any relationship properties defined in the EOModel. For example, an Employee object might have a reference to the employee's department, which in database terms represents a join between the EMPLOYEE and DEPARTMENT tables. Class properties that are relationships are represented in the object graph as faults until they're accessed.

Uniquing, Snapshots, and Faults

When you fetch objects in an Enterprise Objects Framework application, the Framework has mechanisms for ensuring that the integrity of the fetched data is maintained. To this end, the Framework implements these features:

- Uniquing

  Enterprise Objects Framework maintains the mapping of each enterprise object to its corresponding database row, and uses this information to ensure that your object graph does not have two (possibly inconsistent) objects for the same database row.

- Snapshots

  When objects are fetched, Enterprise Objects Framework records the state of the corresponding database row. This information is used when changes are saved back out to the database to ensure that the row data has not been changed by someone else since it was last fetched. The information is also used to only update attributes that have changed, rather than all of them.

- Faults

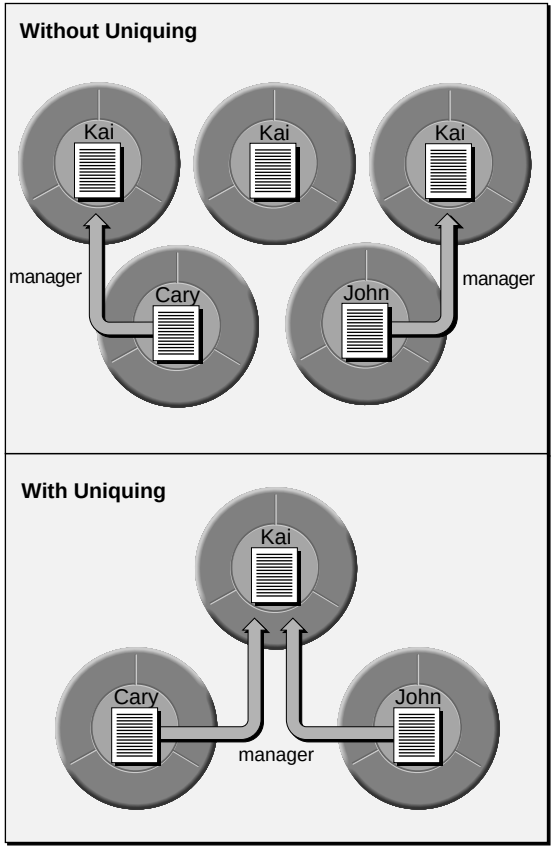
  The objects at the destination of a fetched object's relationships are only fetched on demand; however, these objects are represented in your application by stand-in objects called *faults* to make retrieval of the actual objects easier.

These topics are discussed in more detail in the following sections.

Uniquing

In marrying relational databases to object-oriented programming, one of the key requirements is that a row in the database be associated with only one enterprise object in a given context in your application. Uniquing of enterprise objects limits memory usage and allows you to know with confidence that the object you're interacting with represents the true state of its associated row as it was last fetched into the object graph.

Without uniquing, you'd get a new enterprise object every time you fetch its corresponding row, whether explicitly or through resolution of relationships. This is illustrated in Figure 49.



This shows the enterprise objects that would exist after fetching three employee objects without uniquing. Kai is Cary's and John's manager. On fetching an object for Cary, an object representing Kai is created to resolve the manager relationship. If you then fetch an object for Kai, a separate object is created. Fetching an object for John then causes yet another object representing Kai to be created. Kai's row in the database can be altered between any of these individual fetches, resulting in objects representing the same row, but with different data.

Using uniquing results in only one object ever being created for Kai. In this case, even though Kai's row can be changed, your application has a single view of Kai's data. The data may not reflect what's in the database if another user changes it, but there's no ambiguity within your application.

Figure 49. Uniquing of Enterprise Objects

Uniquing occurs in the control layer, and it's based on an object's globalID. A globalID consists of an object's primary key and its associated entity. When a row is fetched to create an object in a particular EOEditingContext, its globalID is checked against the objects already in the EOEditingContext. If a match is found, the newly fetched object isn't added to the context.

A single enterprise object instance exists in one and only one EOEditingContext, but multiple copies of an object can exist in different editing contexts. In other words, object uniquing is scoped to a particular editing context.

Snapshots

When an EODatabaseContext fetches objects from the database, it asks its EODatabase to record a *snapshot* of the state of the corresponding database row.

A snapshot is a dictionary object recording a row's primary key, class properties, foreign keys for class property relationships, and the attributes of that object that are used for locking during an update. (Primary keys and attributes used for locking are defined in a model; see the book *Enterprise Objects Framework Tools and Techniques*.) A snapshot is recorded under the globalID of its enterprise object whenever the object is fetched or modified.

When changes to an object are saved to the database, the snapshot is compared with the corresponding database row to ensure that the row data hasn't changed since the object was last fetched. For a discussion of how this relates to the update strategy you set for your application, see the section "Locking and Update Strategies" on page 217.

For more information on snapshots, see the EODatabaseContext class specification in the *Enterprise Objects Framework Reference*.

Faults

One of the most powerful and useful features of the Framework's database level is that it automatically resolves the relationships defined in a model. It does so by delaying the actual retrieval of data—and communication with the database—until the data is needed. This delayed resolution of relationships occurs in two stages: the creation of a placeholder object for the data to be fetched, and the fetching of that data only when it's needed.

When the database level fetches an object, it examines the relationships defined in the model and creates objects representing the destinations of the fetched object's relationships. For example, if you fetch an employee object, you can ask for its manager and immediately receive an object; you don't have to get the manager's employee ID from the object you just fetched and fetch the manager yourself.

The database level doesn't immediately fetch data for the destination objects of relationships, however. Fetching is fairly expensive, and

further, if the database level fetched objects related to the one explicitly asked for, it would also have to fetch the objects related to those, and so on, until all of the interrelated rows in the database had been retrieved. To avoid this waste of time and resources, the destination objects created are stand-ins, or *faults*.

Faults come in two varieties: single-object faults for to-one relationships, and array faults for to-many relationships. In Java, a single-object fault is merely a partially initialized enterprise object. It's been created with a constructor of the form:

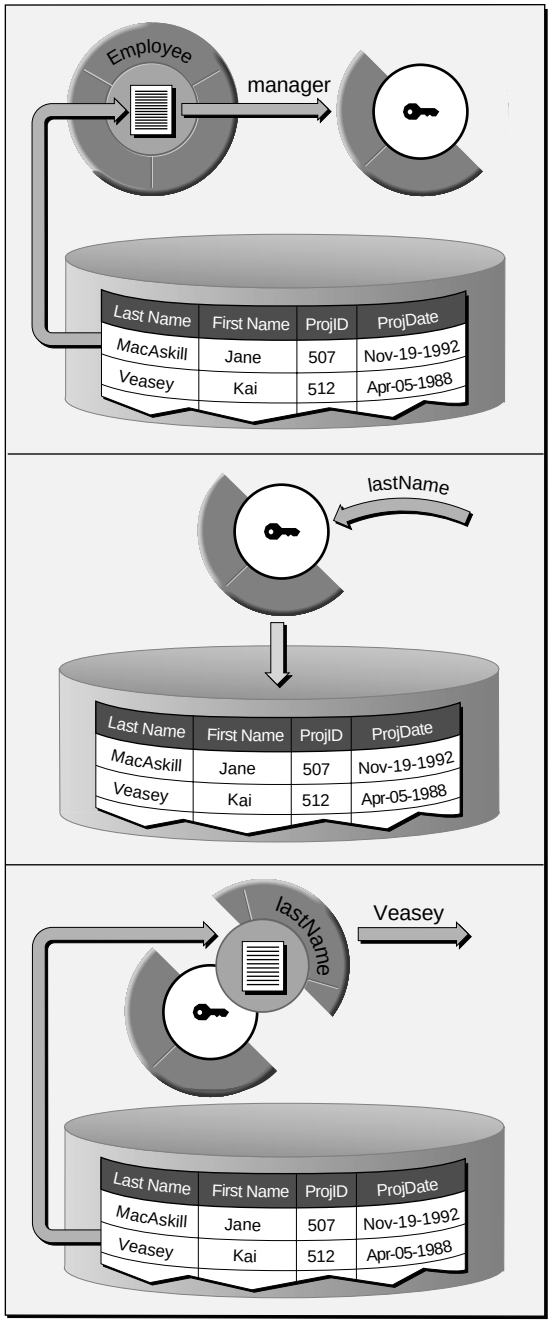
```
public MyCustomEO (
  EOEditingContext anEOEditingContext,
  EOClassDescription anEOClassDescription,
  EOGlobalID anEOGlobalID)
```

and so is already associated with a particular editing context, a class description, and a globalID. However, the object's data hasn't yet been fetched from the database. This part of the object's initialization is delayed until the object receives a message which requires it to fetch its data.

In Objective-C on the other hand, single-object faults are objects of a special class (EOFault) whose instances transform themselves into actual enterprise objects—and fetch their data—the first time they're accessed. These Objective-C faults occupies the same amount of memory as an instance of the target class (into which it's eventually transformed), and stores the information needed to retrieve the data associated with the fault (the source globalID and relationship name). A fault object thus consumes about as much memory as an empty instance of its target class.

An Objective-C fault behaves in every way possible as an instance of its target class until it receives a message it can't cover for. For example, if you fetch an Employee object and ask for its manager, you get a fault object representing another Employee object. If you send a **class** message to this fault object, it returns the Employee class. If you send the fault object a message requesting the value of an attribute, such as **lastName**, however, it uses the EODatabaseContext that created it to retrieve its data from the database, overwrites its class identity, and invokes the target class's implementation of **lastName**.

Figure 50 illustrates this process.



The Employee Object "Jane MacAskill" is fetched from the database. Instead of fetching the data for Jane's manager (Kai Veasey) right away, the Framework creates a fault containing the value of the foreign key for Jane's manager relationship. The graphic for the fault has an empty center with a key in it, indicating that it contains no real values yet. The bottom half of the object shows the messages the fault can respond to without first having to fetch its data.

The fault receives a message it can't cover for (lastName).

The fault fetches its data from the database and invokes its lastName method.

The string "Veasey" is returned.

Figure 50. Resolution of a Fault Object

Array faults are treated similarly by both languages. They behave as instances of the NSMutableArray class, and are triggered to fetch their objects by any request for a member object or for the number of objects in the array (the number of objects for a to-many relationship can't be determined without actually fetching them all).

For more information on faults, see the EOFaulting interface specification (Java only), the EOFault class specification (Objective-C only), and the EOFaultHandler class specification (or both languages) in the *Enterprise Objects Framework Reference*.

Uniquing and Faults

When an EODatabaseChannel constructs a fault for a to-one relationship, it checks the globalID for the destination to see whether that object already exists in the EOEditingContext. If so, it simply uses that object to immediately resolve the relationship. This preserves the uniqueness requirement for enterprise objects, in that there's never more than one **globalID** representing the same row in the database. Whether that **globalID** represents an actual enterprise object or a fault doesn't matter, since the data will be fetched when it's needed.

Similarly, if an EODatabaseChannel fetches data for an object that's already been created as a fault, the EODatabaseChannel *fires* the fault. In Java, this simply means that it finishes initializing the object with the data it's fetched. In Objective-C, this means that the database channel turns the fault into an instance of its target class, *without changing its id*, and then initializes the resulting enterprise object. In either case, the process is essentially the same whether you fetch the fault's data or whether the fault fetches the data itself upon being sent a message.

How Changes are Distributed and Applied

An EOEditingContext is responsible for managing the changes that occur to the objects in its object graph. For example, suppose the user edits a value in the user interface in an Application Kit application. This causes the sequence of events illustrated in Figure 51.

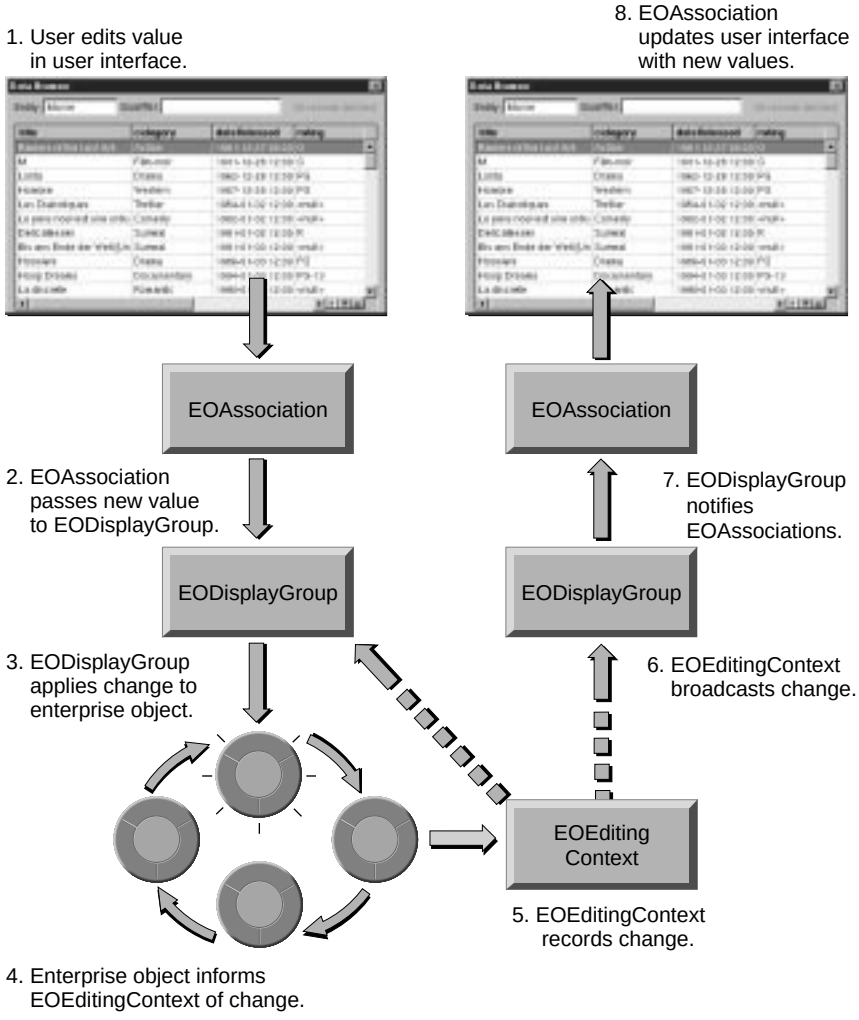


Figure 51. Flow of Events When a User Edits Data

When a user edits data in the user interface:

1. The EOAssociation passes the new value to its EODisplayGroup.
2. The display group applies the changes to the affected enterprise object.
3. The enterprise object notifies the EOEditingContext that it has changed. Specifically, the enterprise object invokes its **willChange** method, which in turn invokes the editing context's **objectWillChange** method (**objectWillChange:** in Objective-C).
4. The editing context records the object in its list of unprocessed changes. (How the editing context manages these changes is described in "How an EOEditingContext Manages Changes to Its Objects" on page 210.)

Then, at the end of the event loop:

5. The editing context records undos.
6. The editing context broadcasts an ObjectsChangedInStoreNotification and an ObjectsChangedInEditingContextNotification (EOObjectsChangedInStoreNotification and EOObjectsChangedInEditingContextNotification in Objective-C).
7. The display group, which is registered to observe the ObjectsChangedInEditingContextNotification, receives the notification and updates the user interface.
8. All views of the data in the application refresh themselves to reflect the change.

Customizing Framework Behavior

During this process, you can customize the behavior of the EOEditingContext by registering for the following notifications and taking the appropriate action.

EOEditingContext Notifications

Notification	Description
ObjectsChangedInStoreNotification (EOObjectsChangedInStoreNotification in Objective-C)	This notification is broadcast whenever objectWillChange observer notifications are processed, which is usually at the end of the event in which the changes occurred.
ObjectsChangedInEditingContextNotification (EOObjectsChangedInEditingContextNotification in Objective-C)	This notification is broadcast whenever changes are made in an EOEditingContext. It's similar to EOObjectsChangedInStoreNotification, except that it contains objects rather than globalIDs. EODisplayGroups listen for this notification to redisplay their contents.

How an EOEditingContext Manages Changes to Its Objects

From the standpoint of an EOEditingContext, the changes you make to objects in an application fall into one of three categories:

- Insertion of a new object
- Deletion of an existing object
- Modification (updating) of an existing object

Normally, when an editing context's objects change (for example, when they're deleted or their data is modified), the processing of changes is deferred until the end of the current event. In the meantime, the editing context buffers pending insertions, deletions, and updates as unprocessed changes.

Note: When a source (master) object has an owning relationship to a destination object (as determined from the EOClassDescription) and the destination object is removed from the master, the destination object is marked for deletion from the EOEditingContext. For example, if a purchase order owns a line item and the line item is removed from the purchase order, the line item is marked for deletion from the editing context since the owning relationship implies that a line item can't exist without a purchase order.

When an EOEditingContext processes changes, typically at the end of an event, it does the following to the objects in its unprocessed changes list:

1. Processes deleted objects.

   For a more detailed description of what this entails, see the following section, "How Deleted Objects are Processed."

2. Moves each object to the context's inserted, deleted, or updated list, as appropriate.

3. Snapshots the objects for undo.

4. Posts ObjectsChangedInStoreNotification and ObjectsChangedInEditingContextNotification.

How Deleted Objects are Processed

Just like inserted and updated objects, deleted objects are normally processed at the end of the event in which the change was made. However, you can use the method **setPropagatesDeletesAtEndOfEvent** (**setPropagatesDeletesAtEndOfEvent:** in Objective-C) to change this behavior so that the editing context only processes deletions right before you save to the database.

The processing of deleted objects entails these steps:

1. Deletions are propagated by sending each object a **propagateDeleteWithEditingContext** message (**propagateDeleteWithEditingContext:** in Objective-C), which then invokes the EOClassDescription method **propagateDeleteForObject** (**propagateDeleteForObject:editingContext:** in Objective-C).

By default, this method applies the delete rule of every relationship (Deny, Nullify, Cascade) to the source object's child objects.

2. The deletion is validated by sending each object the message **validateForDelete**.

   By default, each object forwards this message to its EOClassDescription. Based on the result, the operation is either allowed or refused. For example, referential integrity constraints in your model might state that you can't delete a Department object that still has employees. If a user attempts to delete a department that has employees, the deletion is refused. An enterprise object class can also implement its own version of **validateForDelete** to do some additional processing before passing the check on to its EOClassDescription. For more discussion of validation, see the chapter "Designing Enterprise Objects" on page 65.

Instead of waiting until the end of the event, you can force the processing of inserted, updated, and deleted objects by invoking the EOEditingContext method **processRecentChanges**. EOEditingContext invokes this method on itself before performing certain operations such as **saveChanges**. The sequence of events that occurs when an editing context receives the message **saveChanges** is described in the next section.

Saving Changes

In a running application, the changes users make to objects are reflected in the object graph managed by an EOEditingContext and in the user interface. However, the database isn't updated to reflect these changes until there is an explicit request (typically issued by the user) to save. The sequence of events in a save operation is illustrated in Figure 52.

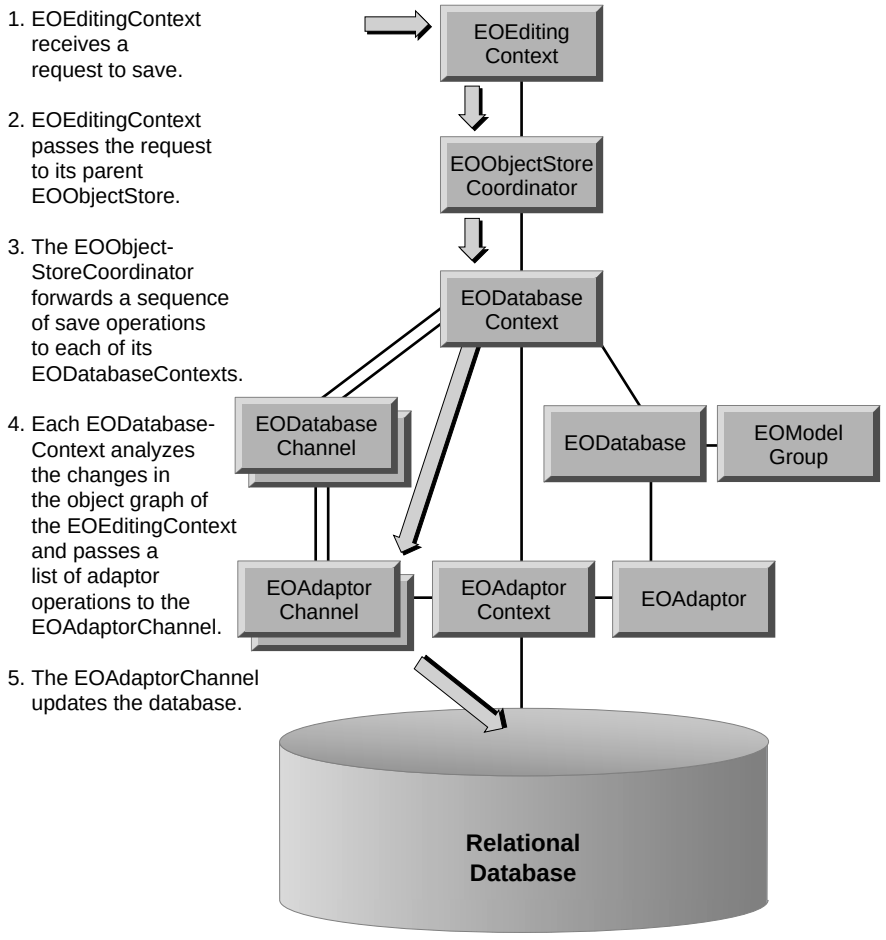


Figure 52. Saving to the Database

When an EOEditingContext receives a request to save in the form of a **saveChanges** message, the following sequence of events occurs:

1. The editing context sends it editors and delegates the message **editingContextWillSaveChanges**.
2. The editing context processes, propagates, and validates deletes.
3. The editing context processes and validates changes for saving.

4. The editing context commits the changes made to its objects to its parent object store by sending the parent the message **saveChangesInEditingContext** (**saveChangesInEditingContext:** in Objective-C). If the editing context is not nested, its parent is typically an EOObjectStoreCoordinator. When an EOObjectStoreCoordinator receives this message, it guides its EOCooperatingObjectStores through a multi-pass save protocol in which each cooperating store saves its own changes and forwards remaining changes to other cooperating stores.

5. After it receives the message **saveChangesInEditingContext**, the object store coordinator sends each of its cooperating stores a **prepareForSaveWithCoordinator** message (**prepareForSaveWithCoordinator:editingContext:** in Objective-C), which informs them that a multi-pass save operation is beginning. When the cooperating store is an EODatabaseContext, it takes this opportunity to generate primary keys for any new objects in the editing context.

6. The coordinator sends each of its cooperating stores the message **recordChangesInEditingContext**, which prompts them to examine the changed objects in the editing context, record any operations that need to be performed, and notify the coordinator of any changes that need to be forwarded to other cooperating stores. For example, if in its **recordChangesInEditingContext** method one cooperating store notices the removal of an object from an "owning" relationship but that object belongs to another cooperating store, it informs the other store by sending the coordinator a **forwardUpdateForObject** message (**forwardUpdateForObject:changes:** in Objective-C).

7. The coordinator sends each of its cooperating stores the message **performChanges**. This tells the stores to transmit their changes to their underlying databases. When the cooperating store is an EODatabaseContext, it responds to this message by taking the EODatabaseOperations that were constructed in the previous step, constructing EOAdaptorOperations from them, and giving the EOAdaptorOperations to an available EOAdaptorChannel for execution.

8. If **performChanges** fails for any of the EOCooperatingObjectStores, all stores are sent the message **rollbackChanges**.

9. If **performChanges** succeeds for all EOCooperatingObjectStores, the receiver sends them the message **commitChanges**, which has the effect of telling the adaptor to commit the changes.

10. If **commitChanges** fails for a particular cooperating store, that store and all subsequent ones are sent the message **rollbackChanges**. However, the stores that have already committed their changes do not roll back. In other words, the EOObjectStoreCoordinator doesn't perform the two-phase commit protocol necessary to guarantee consistent distributed update.

11. If the save operation was successful, the editing context updates its object snapshots.

12. Once it has committed its changes to its parent object store, the editing context posts the EditingContextDidSaveChangesNotification (EOEditingContextDidSaveChangesNotification in Objective-C).

Customizing Framework Behavior

You can customize the behavior a save operation by assigning delegates to EOEditingContext and EOAdaptorChannel, and implementing the any of the following delegate methods.

EOEditingContext Delegate Methods

Java Method	Objective-C Method	Description
editingContext ShouldValidateChanges	editingContext ShouldValidateChanges:	This method is invoked when an EOEditingContext receives a saveChanges message. If the delegate returns **false** (NO), changes are saved without first performing validation. You can use this method to provide your own validation mechanism.

EOEditingContext Delegate Methods (Continued)

Java Method	Objective-C Method	Description
editingContext WillSaveChanges	editingContext WillSaveChanges:	This method is invoked when an EOEditingContext receives a saveChanges message. You can use this method to perform other pre-save validation.

EOAdaptorChannel Delegate Methods

Java Method	Java Method	Description
databaseContextWillOrder AdaptorOperations	databaseContext: willOrder AdaptorOperations FromDatabaseOperations:	This method is invoked when EODatabaseContext receives a performChanges message. You can use this method to construct your own adaptor operations, for instance, possibly transform a delete operation into an update, or a stored procedure invocation.
databaseContext WillPerform AdaptorOperations	databaseContext: willPerform AdaptorOperations: adaptorChannel:	This method is invoked from the EODatabaseContext performChanges method. This method is useful for applications that need a special ordering of adaptor operations; for example, to avoid violating any database referential integrity constraints.

You can also register to receive the notifications listed below

EOEditingContext Notifications

Notification	Description
EditingContextDidSaveChangesNotification (EOEditingContextDidSaveChangesNotification in Objective-C)	This notification is broadcast after changes are saved to the editing context's parent object store.
ObjectsChangedInStoreNotification (EOObjectsChangedInStoreNotification in Objective-C)	This notification is broadcast by the database context when object updates are committed to the database.

Locking and Update Strategies

An update operation includes the following ingredients:

- An enterprise object whose data values have been changed
- A means of identifying the row in the database that corresponds to the object
- A strategy for handling update conflicts—either by preventing them from occurring, or by detecting and handling them when they do occur.

There must also be a transaction in progress.

The "means of identifying" a row is the primary key or global ID.

An *update strategy* determines how updates should be made in the face of changes by others. For example, one strategy is to lock a row when it is read so that no one else can change it until you're done with it; this is called *pessimistic locking*. Another strategy is to compare the state of a row as you fetched it—that is, the row's snapshot—with the database row at update time to confirm that the database row hasn't been changed by someone else. This is called *optimistic locking*, because it assumes a conflicting update won't occur, but does check at the last minute. You can set your update strategy using the EODatabaseContext method **setUpdateStrategy** (**setUpdateStrategy:** in Objective-C). Optimistic locking is the default.

Enterprise Objects Framework also supports "on-demand" locking, in which specific optimistic locks can be promoted to database locks during the course of program execution. In other words, you can lock single objects. There are three ways to use on-demand locking. Use the EODatabaseContext method **lockObjectWithGlobalID** (**lockObjectWithGlobalID:editingContext:** in Objective-C) to lock a database row for a particular object. Use the EODatabaseContext method **objectsWithFetchSpecification** (**objectsWithFetchSpecification:editingContext:** in Objective-C) with a fetch specification that's configured to lock rows as they're fetched. Or use the EOEditingContext method **lockObject** (**lockObject:** in Objective-C).

Handling Conflicts

The locking approach you use determines at what point conflicts are detected and how you can handle them.

- Pessimistic locking

  When you use pessimistic locking, conflicts are detected as soon as you fetch a row. This is because when you fetch a row with pessimistic locking, you attempt to put a lock on it. If someone else has a lock on the row, the lock (and hence, the fetch operation) is refused. Your application can display a panel at that point telling the user to try again later.

  Since pessimistic locking puts a lock on a row when it fetches it, you can generally assume that you won't experience conflicts when you save changes. However, this behavior is ultimately dependent on how the database server handles locks.

- Optimistic locking

  When you use optimistic locking, conflicts aren't detected until you attempt to save. At that point, the database row is checked against the snapshot to make sure the row hasn't changed. If the row and the snapshot don't match, the save operation is aborted, the transaction is rolled back, and an exception is thrown. To handle the error you can catch the exception, refresh the conflicted object from the updated database data, and save again.

- On-demand locking

  On-demand locking mixes characteristics of both pessimistic and optimistic locking. With on-demand locking, you've already fetched the object, and you're trying to get a lock on it after the fact. When you try to get a lock on the object's corresponding database row, you can get a failure for one of two reasons: either because the row doesn't match the snapshot (optimistic locking), or because someone else has a lock on the row on the server (pessimistic locking).

When on-demand locking fails for either reason, it throws an exception. To handle the error you can catch the exception, refresh the conflicted object from the updated database data, and try to get a lock on it again.

As with pessimistic locking, because on-demand locking locks the row, you can generally assume that you won't experience conflicts when you save changes. Again, this behavior is ultimately dependent on how the database server handles locks.

Transactions

For the most part, Enterprise Objects Framework handles transactions for you. You rarely (if ever) need to explicitly start or end a transaction yourself—it generally happens as the by-product of another operation, such as a fetch or save. However, understanding how transactions are handled in the Framework can help you to make the right decisions for your application.

How transactions are handled in the Framework depends on the locking mode you have set. There are three different possibilities:

- Optimistic locking
- Pessimistic locking
- "On-demand" locking of individual objects

For a detailed description of these locking modes, see the section "Locking and Update Strategies" on page 217.

The way that each of these modes affects transactions is described in the following sections.

Transactions and Optimistic Locking

If you're using optimistic locking (the default) and you're just fetching objects, Enterprise Objects Framework never explicitly starts or stops transactions. Instead, when a SELECT is performed on a database row, opening (and subsequently closing) a transaction is typically handled by the database server itself, implicitly. Ultimately, it is the responsibility of the adaptor for each database server to ensure that the right thing happens.

Under optimistic locking, Enterprise Objects Framework explicitly starts a transaction when you perform a save operation. A save operation consists of three basic parts:

- Beginning a transaction
- Performing the specified operations (including checking snapshots)
- Committing the transaction, or rolling back if the transaction fails. In either case, the transaction is closed.

Transactions and Pessimistic Locking

When you use pessimistic locking, Enterprise Objects Framework explicitly starts a transaction as soon as you fetch objects, and every object you fetch is locked. The transaction stays open until you commit it (using the EOEditingContext method **saveChanges**), or roll it back (using the EOEditingContext method **invalidateAllObjects**).

Consequently, using pessimistic locking is very expensive. It's not suitable for applications that have user interaction since large portions of your database could be locked down for indeterminate periods of time. A good alternative to pessimistic locking is using on-demand locking to lock individual objects.

Transactions and On-Demand Locking

When you use on-demand locking to get a server lock on an object, Enterprise Objects Framework explicitly opens a transaction and keeps it open as long as you have a lock on the object. The transaction stays open until you commit it (using the EOEditingContext method **saveChanges**), or roll it back (using the EOEditingContext method **invalidateAllObjects**).

Chapter 8

Answers to Common Design Questions

This chapter answers questions to common application and framework design questions. For a discussion of design issues affecting enterprise objects, see the chapters "Designing Enterprise Objects" on page 65 and "Advanced Enterprise Object Modeling" on page 105.

The topics covered in this chapter are as follows:

How Can I Improve Performance?

In an Enterprise Objects Framework application, every trip to the database and every object fetched is a potential drag on performance. Consequently, a large part of designing for performance entails answering these questions:

- How can I minimize my application's trips to the database?
- When I *do* have to make trips to the database, how can I best take advantage of them?
- How can I avoid fetching objects I'll never need, while still maintaining access to objects I might need?

Enterprise Objects Framework has several built-in features for intelligently managing your application's interactions with the database. It also has hooks for fine-tuning this behavior to get the best performance for your application.

Controlling the Number of Objects Fetched

If you define a fetch specification in your model, you can set a fetch limit for it in EOModeler. You can also define what should happen if the fetch limit is reached. For more information, see *Enterprise Objects Framework Tools and Techniques*.

If you're not using a predefined fetch specification, you set the fetch limit programmatically using EOFetchSpecification's **setFetchLimit** method (**setFetchLimit:** in Objective-C) passing an integer value indicating the maximum number of objects to fetch (an unsigned integer value in Objective-C). The default value is zero, indicating no fetch limit.

The EODatabaseContext will either stop fetching objects when this limit is reached or ask the EOEditingContext's message handler to ask the user whether it should continue fetching. The default behavior simply stops fetching, so if you want to prompt the user, send **setPromptsAfterFetchLimit** (**setPromptsAfterFetchLimit:** in Objective-C) to the fetch specification with **true** (YES in Objective-C) as the argument. For more information on managing fetch limits, see the EOFetchSpecification class description and the EOEditingContext EOMessageHandlers interface description in the *Enterprise Objects Framework Reference*.

Faulting

When an EODatabaseContext fetches an object, it uses the relationships defined in the model to fetch related objects. For example, if you fetch an employee object, you can access its manager directly; you don't have to get the manager's employee ID from the object you just fetched and fetch the manager yourself.

However, EODatabaseContext doesn't fetch related objects immediately, since they may never be accessed and fetching can be expensive. Instead the destination objects created are stand-ins, called *faults*, that fetch their data the first time they're accessed.

When a fault is accessed (sent a message for which it must get its data to respond), it triggers its EODatabaseContext to fetch its data and finish initializing it. This works well for limited numbers of objects. However, suppose you fetch multiple employees and then want to retrieve each employee's department. You'd have to loop over all of the employees and fetch each employee's department fault individually, resulting in numerous trips to the database.

To avoid these unnecessarily trips to the database, you can fine-tune faulting behavior for additional performance gains by using two different mechanisms: batch faulting, and prefetching relationships.

Batch Faulting

When you access a fault, its data is fetched from the database. However, triggering one fault has no effect on other faults—it just fetches the object or array of objects for the one fault. You can take advantage of this expensive round trip to the database server by batching faults together. When you do this, triggering one fault (such as an employee's department) has the effect of fetching multiple faults. This reduces the number of fetches—the next time you access an employee's department, it's less likely to require a trip to the database.

You can configure batch faulting in a model with EOModeler. With this approach, you specify the number of faults for the same entity or relationship that should be triggered along with the first fault. For more information on setting batch faulting in an EOModel, see *Enterprise Objects Framework Tools and Techniques*.

To actually control which faults are triggered along with the first one, you can use the EODatabaseContext method **batchFetchRelationship** (**batchFetchRelationship:forSourceObjects:editingContext:** in Objective-C). For example, given an array of Employee objects, this method can fetch all of their departments with one round trip to the server, rather than asking the server for each of the employee's departments individually. For more information, see the EOFetchSpecification class description in the *Enterprise Objects Framework Reference*.

Prefetching Relationships

Sometimes it's more efficient to specify *prefetching relationships* so that related objects are fetched at the same time. For example, when fetching employees, you can define a prefetching relationship between an employee and a department to force these objects to be fetched as well, as opposed to having faults created for them. Although prefetching increases the initial fetch cost, it can improve overall performance by reducing the number of round trips made to the database server.

If you define your fetch specification in a model, you can configure its prefetching behavior in EOModeler. For more information, see *Enterprise Objects Framework Tools and Techniques*.

Alternatively, you can programmatically set prefetching relationships by sending **setPrefetchingRelationshipKeyPaths** (**setPrefetchingRelationshipKeyPaths:** in Objective-C) to an EOFetchSpecification object and passing an array of relationship key paths whose destinations should be fetched along with the objects specified. For more information, see the EOFetchSpecification class description in the *Enterprise Objects Framework Reference*.

Caching an Entity's Objects

You can cache an entity's objects in memory for quick access. Caching an entity's objects allows Enterprise Objects Framework to evaluate queries in memory, thereby avoiding round trips to the database. This is most useful for read-only entities, where there is no danger of the cached data getting out of sync with database data. This technique should only be used with small tables, since it fetches the entire table into memory.

To set up object caching on an entity, you can use the Advanced Entity Inspector in EOModeler or you can do it programmatically using the EOEntity method **setCachesObjects** (**setCachesObjects:** in Objective-C). For more information on configuring object caching in EOModeler, see *Enterprise Objects Framework Tools and Techniques*, and for more information on object caching, see the EOEntity class specification in the *Enterprise Objects Framework Reference*.

Creating an EOModel for Optimal Performance

The way you design your EOModel has a direct effect on how your application interacts with the database, and consequently, on performance. There are a few general guidelines you should observe:

- Avoid flattening objects whenever possible.
- Use inheritance wisely.
- Don't set BLOB attributes to be used for locking.

Each is discussed in the following sections.

Avoid Flattening Attributes

Flattening attributes has two major drawbacks:

1. The values of flattened attributes can get out of sync with the object graph (which represents the most current view of data in your application). This limitation doesn't apply if you're flattening a one-to-one relationship in order to map a class across multiple tables.

2. Fetching objects that span multiple database tables requires database joins, which are expensive. If you find yourself designing an application that requires flattened attributes, you should consider whether there's a more efficient approach.

Instead of flattening attributes, you can directly traverse relationships in the object graph. For example, the following statements access the value of a **departmentName** property belonging to the Department object to which Employee has a relationship:

In Java:

```
// Get the name of the Employee's department
employee.department().departmentName();

// Set the name of the employee's department
employee.department().setDepartmentName(newName);
```

In Objective-C:

```
// Get the name of the Employee's department
[[employee department] departmentName];

// Set the name of the employee's department
[[employee department] setDepartmentName:newName];
```

For more discussion of this subject, see the chapter "Designing Enterprise Objects" on page 65.

Use Inheritance Wisely

As discussed in the chapter "Designing Enterprise Objects," the way that you map an object hierarchy onto a relational database in your EOModel can have a significant effect on performance. You should observe the following guidelines:

- Avoid mapping a deep object hierarchy onto a relational database since it will probably result in multiple fetches and joins.
- Try to avoid using vertical inheritance mapping, since it's the least efficient of the possible approaches.

Don't Use BLOB Attributes For Locking

In EOModeler the Used For Locking setting indicates whether an attribute should be checked for changes before an update is allowed. This setting applies when you're using Enterprise Object Framework's default update strategy, optimistic locking. Under optimistic locking, the state of a row is saved as a *snapshot* when you fetch it from the database. When you perform an update, the snapshot is checked against the row to make sure the row hasn't changed. If you set Used For Locking for an attribute whose data is a BLOB type, it can increase the cost of updating the row containing the BLOB.

Ideally, you should store BLOBs in their own table away from more commonly accessed attributes.

Updating the User Interface Display

When objects change in the EOEditingContext for an EODisplayGroup, the EODisplayGroup by default refreshes all of its EOAssociations, even

if none of the EODisplayGroup's objects is in the EOEditingContext notification change list.

This "universal" refresh is sometimes necessary because EOAssociations may display derived values (through key paths or business methods) that depend on objects other than the ones being displayed. However, if you know that your user interface doesn't display derived data, you can specify that an EODisplayGroup's EOAssociation objects be refreshed only if the EODisplayGroup objects change.

There are different ways to accomplish this:

- In Interface Builder, display the Attributes view of the EODisplayGroup Inspector and uncheck "Refresh All".
- In your code, include a statement such as the following:

  In Java:

  ```
  myDisplayGroup.setUsesOptimisticRefresh(true);
  ```

  In Objective-C:

  ```
  [myDisplayGroup setUsesOptimisticRefresh:YES];
  ```

  This is equivalent to unchecking "Refresh All" in Interface Builder for **myDisplayGroup**.
- Implement the EODisplayGroup delegate method **displayGroupShouldRedisplay** (**displayGroup:shouldRedisplayForChangesInEditingContext:** in Objective-C) to control when redisplay occurs.

How Do I Generate Primary Keys?

Enterprise Objects Framework requires you to specify a primary key for each entity in a model. In applications that create new enterprise objects to insert into a database, unique values must be generated and assigned to an object's primary key. The Framework typically does this for you, but you can override or customize its default behavior.

Note: Enterprise Objects Framework doesn't support modifiable primary key values—you shouldn't design your application so that users can change a primary key's value. If you really need this behavior, you have to implement it by deleting an affected object and reinserting it with a new primary key.

Defining a Primary Key

When designing a database, keep the following tips in mind for defining primary keys:

- Don't use floating point values such as doubles and dates because they aren't precise in equality tests.
- Use integer or 12 byte binary primary keys when you want Enterprise Objects Framework to generate primary key values automatically. For more information on the format of 12 byte primary keys, see the constructor description (or the method description for **assignGloballyUniqueBytes:** in Objective-C) in the EOTemporaryGlobalID class specification in the *Enterprise Objects Framework Reference*.
- Try to avoid using compound keys. A compound key incurs additional overhead in not only its entity but also in related entities: the destination entities of all to-one relationships must contain an attribute for each primary key attribute in the source. In addition, you can't use Enterprise Objects Framework's automatic primary key generation mechanism for compound primary keys.
- You can improve the efficiency of enterprise object inheritance support by encoding the class of an object in its primary key. When the class of an object is encoded in its key and you implement the EOModelGroup delegate method to tell the Framework the subentity and subclass for a key, Enterprise Objects Framework creates a more efficient fault for the object than it would otherwise. Try to encode the class of an object in a large integer or binary key instead of using a compound key. For more information, see the section "Delegation Hooks for Optimizing Inheritance" on page 134 in the "Advanced Enterprise Object Modeling" chapter.

Generating Primary Key Values

There are four ways to provide primary key values for enterprise objects:

1. *An enterprise object can provide its own primary key value.* With this approach, the primary key must be a class property of the object. If the primary key value of an object is **null** (**nil** in Objective-C) or zero when the Framework attempts to insert it, the Framework falls back on one of the other mechanisms to provide the value.

2. *An EODatabaseContext's delegate provides a primary key value.* If the EODatabaseContext that's inserting an enterprise object has a delegate, and if the delegate has a method called **databaseContextNewPrimaryKey** (**databaseContext:newPrimaryKeyForObject:entity:** in Objective-C) that returns a non-**null** (non-**nil**) value, the Framework uses the returned object as the primary key value.

3. *A database stored procedure provides a primary key value.* If an enterprise object's entity has a stored procedure assigned to the NextPrimaryKeyProcedureOperation, the Framework invokes the stored procedure and uses the result as the primary key value.

4. *Your adaptor provides a primary key value using a database-specific mechanism.* Each adaptor provides a database-specific implementation of the method **primaryKeyForNewRowWithEntity** (**primaryKeyForNewRowWithEntity:** in Objective-C) that provides unique values for primary key attributes.

   This is the technique used when your primary keys are integers. However, as described in the preceding section, when you want Enterprise Objects Framework to generate primary keys, you can also use 12 byte NSDatas. The difference is that integer primary keys are fetched from the database, whereas NSData keys are generated on the client (see the EOTemporaryGlobalID class specification in the *Enterprise Objects Framework Reference* for more information). Consequently, using 12 byte NSDatas is faster, but integer primary keys have the advantage of being more readable.

If the Framework can't assign a primary key using one of the mechanisms above, it throws an exception.

The following sections provide more information on when and how to use each mechanism.

When the Enterprise Object Provides the Key

An enterprise object generally provides its own primary key value when the primary key is meaningful to users—a social security number, account number, or part number, for example. In some cases, the user provides the primary key value by entering it in the user interface. In other cases, the enterprise object generates its own unique primary key value. For example, a Part object's primary key could encode the part's type, the plant from which it came, and the batch in which it was made. Although generated, part numbers may still be meaningful to users if they use them to identify parts.

To specify that an enterprise object provides its own key, you must set the primary key attributes as class properties in the object's entity. Your enterprise object class should provide an instance variable or accessor methods for each of the primary key attributes. If you want to provide the primary key value for a newly created enterprise object, be sure to assign it before the object is saved.

Note: In the case of Number objects (NSNumbers in Objective-C), don't set the value to zero unless you intend to have the primary key generated. See the section "Why is EOF Generating Primary Key Values for Number Objects Set to Zero?" on page 237 below for details.)

If an enterprise object generates its own primary key value programmatically, you must generate and assign it in an appropriate method. You could, for example, provide a primary key value when the object is first instantiated by implementing the method **awakeFromInsertion** (**awakeFromInsertionInEditingContext:** in Objective-C).

On the other hand, if your application's user interface provides a way for the user to enter primary key values, you don't need to handle them any differently than you handle the object's other properties. For example, if an application uses social security numbers as the primary keys for employees, it must provide a way for users to enter them. The interface

layer of the Framework takes care of assigning the user-provided value to the object.

The disadvantage of letting users enter primary key values is that there's a chance for data-entry error and the possibility that the object's primary key will need to be modified later. Since Enterprise Objects Framework doesn't support modifiable primary keys, you have to delete an object and reinsert it with a new primary key value to change its primary key. It's generally better to define a "meaningless" primary key to use instead.

When the EODatabaseContext Delegate Provides the Key

An EODatabaseContext's delegate is given an opportunity to provide a primary key value for enterprise objects that don't already have one. This is the most commonly used mechanism in applications that don't use the adaptor's database-specific primary key generation mechanism. You might use the delegate to provide primary key values when you want to avoid making a trip to the database. For example, you might implement this method to generate globally unique identifiers based on an IP address and a time stamp.

To allow your EODatabaseContext's delegate to provide primary keys, implement the method **databaseContextNewPrimaryKey** (**databaseContext:newPrimaryKeyForObject:entity:** in Objective-C). An EODatabaseContext sends this method to its delegate when a newly inserted enterprise object doesn't have a primary key value. If the delegate is not implemented or returns **null** (**nil**), the EODatabaseContext gets a primary key by invoking a stored procedure or using its adaptor's database-specific mechanism.

When a Database Stored Procedure Provides the Key

You typically use a stored procedure to provide primary key values when you need to override the adaptor's database-specific mechanism but still need to make a trip to the database to generate values.

To use a stored procedure to provide primary key values, you must define the stored procedure in your model. Stored procedures are read from the database when you create a new model and included in the model's **.eomodeld** file. You can also add stored procedures in EOModeler using the Stored Procedure view of the Model Editor.

After defining the stored procedure, you assign it to an entity. You can set it in EOModeler: In the Stored Procedure Inspector, type the name of the stored procedure in the Get PK field. Alternatively, you can set it programmatically using EOEntity's **setStoredProcedure** method (**setStoredProcedure:forOperation:** in Objective-C). For more information on defining stored procedures and assigning them to entities, see the section "How Do I Invoke a Stored Procedure?" on page 242.

When the Adaptor Provides the Key

Each adaptor provides a database-specific mechanism for generating primary keys. Unless you specify one of the other four mechanisms, Enterprise Objects Framework automatically uses the adaptor's mechanism.

Each adaptor provides an implementation of the method **primaryKeyForNewRowWithEntity** (**primaryKeyForNewRowWithEntity:** in Objective-C). When invoked, this method returns a unique primary key value. For example, the Oracle adaptor uses Oracle sequences to generate unique values.

To use the adaptor's database-specific mechanism, you must be sure that your database accommodates the adaptor's scheme. The primary keys of the affected tables must be simple (that is, they can't be compound primary keys), and they must be number types.

To modify your database so that it supports the adaptor's mechanism for generating primary keys:

1. In EOModeler's Model Editor, select the entities for which you want the adaptor to generate primary key values.
2. Choose Property ► Generate SQL.
3. In the SQL Generation panel that appears, check the "Create Primary Key Support" box as well as any of the others that you might need.

The following sections describe the support added to your database for each of Enterprise Objects Framework's adaptors.

Informix and Sybase

The Informix and Sybase adaptor use the same approach to generating primary key values. Both adaptors use a table named eo_sequence_table to keep track of the next available primary key value for a given table. The table contains a row for each table for which the adaptor provides primary key values.

The statements used to create the eo_sequence_tables are:

Informix	Sybase
create table eo_sequence_table (table_name varchar(32, 0), counter integer)	create table eo_sequence_table (table_name varchar(32), counter int null)

The adaptors use a stored procedure called eo_pk_for_table to access and maintain the primary key counters in eo_sequence_table. The stored procedures are defined as follows:

Informix	Sybase
create procedure eo_pk_for_table (tname varchar(32)) returning int; define cntr int; update EO_SEQUENCE_TABLE set COUNTER = COUNTER + 1 where TABLE_NAME = tname; select COUNTER into cntr from EO_SEQUENCE_TABLE where TABLE_NAME = tname; return cntr; end procedure;	create procedure eo_pk_for_table @tname varchar(32) as begin declare @max int update eo_sequence_table set counter = counter + 1 where table_name = @tname select counter from eo_sequence_table where table_name = @tname end

The stored procedures increment the counter in the eo_sequence_table row for the specified table, select the counter value, and return it. The Informix and Sybase adaptor's **primaryKeyForNewRowWithEntity** methods execute the eo_pk_for_table stored procedure and return the stored procedure's return value.

ODBC

The approach taken by the ODBC adaptor is very similar to that of the Informix and Sybase adaptors. The ODBC adaptor uses a table named EO_PK_TABLE to keep track of the next available primary key value for a table, but the ODBC adaptor can create this table on demand. (The Informix and Sybase adaptors do not create the table and corresponding stored procedures. Rather, you create them ahead of time using the SQL Generation panel in EOModeler.)

The ODBC adaptor's **primaryKeyForNewRowWithEntity** method attempts to select a value from the EO_PK_TABLE for the new row's table. If the attempt fails because the table doesn't exist, the adaptor creates the table using the following SQL statement:

```
CREATE TABLE EO_PK_TABLE (
  NAME TEXT_TYPE(40),
  PK NUMBER_TYPE
)
```

where *TEXT_TYPE* is the external (database) type for characters and *NUMBER_TYPE* is the external type for the table's primary key attribute. The ODBC adaptor sets the PK value for each row to the corresponding table's maximum primary key value plus one. After determining a primary key value for the new row, the ODBC adaptor updates the counter in the corresponding row in EO_PK_TABLE.

Oracle

The Oracle adaptor uses sequence objects to provide primary key values. It creates a sequence using the following SQL statement:

```
create sequence table_SEQ
```

where *table* is the name of a table for which the adaptor provides primary key values. The adaptor sets the sequence start value to the corresponding table's maximum primary key value plus one.

Why Can't I Use Identity Columns?

Some databases provide mechanisms that automatically generate primary key values. For example, Sybase allows you to specify *identity* columns that automatically replace nulls with unique values. In databases that don't provide identity columns, you can define *triggers* to produce the

same result. These mechanisms are very useful when users interact directly with the database using SQL. However, they are difficult to use in applications that mediate between users and a database. You shouldn't use them in applications built with Enterprise Objects Framework.

Suppose that a database application allowed you to insert a row without providing a primary key value. An identity column or database trigger could generate an identifying value for the row, but the corresponding application object wouldn't have the value. The application could attempt to fetch the object using the values provided by the user, but a query that doesn't specify a primary key value might return more than one row. As a result, the application can't guarantee that it will be able to associate the current object with a row in the database. For this reason, Enterprise Objects Framework requires that you assign a primary key value to an object before it's inserted in the database.

Why is EOF Generating Primary Key Values for Number Objects Set to Zero?

The EODatabaseContext assumes that an Number object (NSNumber in Objective-C) with a single attribute primary key value set to zero is a newly created instance, and therefore, will attempt to generate the primary key. This behavior allows you to use scalar data types (such as **int**) as an object's primary key, and still rely on automatic primary key generation.

This can cause problems if you have an existing database containing rows that use zero as the primary key value. The EODatabaseContext will incorrectly assume that an object created from that row needs a new primary key. This behavior may result in invalid foreign key references in other tables of your database.

To alter this behavior, assign a delegate to the EODatabaseContext object and implement the **databaseContextNewPrimaryKey** delegate method (**databaseContext:newPrimaryKeyForObject:entity:** in Objective-C) to return a Number object of value zero if the primary key should remain zero (an NSNumber in Objective-C), otherwise return **null** (**nil**). Returning **null** will tell EOF to find another way to generate the primary key value as described above.

Summary

The following table summarizes the primary key generation options you have to choose from.

Mechanism	Primary Use
Object provides its own value	When the primary key value is meaningful to users and is displayed in the application's user interface.
EODatabaseContext delegate method	When you don't want to use the adaptor's mechanism.
Stored procedure	When you want to use your own stored procedure to provide primary key values.
Adaptor's mechanism	When the primary key is a simple (not compound), numeric value that is not meaningful to users.

How Do I Use My Database Server's Integrity-Checking Features?

Most database systems offer features to help you maintain the integrity of your data. You can assign default values to columns, define rules that specify the format or allowable range of a column's values, and define constraints or triggers to enforce relational integrity rules. Enterprise Objects Framework has its own brand of solutions for the same issues. You have to decide whether to use the database system's solution, the Framework's solution, or a combination of the two. The decision involves answering the following questions:

- Can I avoid using the database's integrity-checking features?
- Is it possible that non-Enterprise Objects Framework tools and applications will access the database?

- Can I use the database system's feature without interfering with the way Enterprise Objects Framework works?
- How can I use both the database system's and Enterprise Objects Framework's solutions?

When you implement integrity checking in your Enterprise Objects Framework applications, you can reject erroneous data or illegal operations as soon as a user performs an invalid action. Enterprise Objects Framework relies on application-side integrity checking to provide feedback to users and to handle errors. Without it, it is much more difficult for you to develop the user interfaces for your Enterprise Objects Framework applications.

Because client-side business logic is required to create a highly interactive user interface and because duplication of business logic is inefficient and error-prone, you should try to avoid using database integrity-checking features. Sometimes, however, it's unavoidable. You usually use database integrity checking when users can access a database in many ways (using Enterprise Objects Framework applications, non-Enterprise Objects Framework applications, and interactive SQL sessions, for example). In this case, you may have to use the features of your database server to assure your data's integrity. As a result, you may choose to implement integrity checking in both your Enterprise Objects Framework applications and in the database.

The following sections discuss guidelines for using the integrity-checking features of your database in concert with an Enterprise Objects Framework application.

Defaults

Many databases allow you to specify a default value for a column. When a NULL value is inserted (or updated) in a column with a default, the database substitutes the default value for the NULL.

If you define defaults in your database, you should specify the defaults in your Enterprise Objects Framework application as well. Generally, you assign default values in your enterprise object's **awakeFromInsertion** method (**awakeFromInsertionInEditingContext:** in Objective-C).

For example:

In Java:

```
public void awakeFromInsertion(EOEditingContext ec)
{
  super.awakeFromInsertion(ec);
  // Assign current date to memberSince
  if (memberSince == null)
      memberSince = new NSGregorianDate();
}
```

In Objective-C:

```
- (void)
  awakeFromInsertionInEditingContext:
      (EOEditingContext *)ec
{
  [super awakeFromInsertionInEditingContext:ec];
  // Assign current date to memberSince
  if (!memberSince)
      memberSince = [[NSCalendarDate date] retain];
}
```

An alternative is to fetch newly inserted objects immediately after you save them to the database. If you don't assign the default values before you save an object and you don't refetch the object from the database after you save, the Framework's object snapshots will not be in sync with the contents of the database. As a result, the Framework may prevent subsequent updates to the object.

Rules That Validate Values

Many databases allow you to define a rule (or constraint) for a column. A rule can verify that a value is in a proper format or is within an acceptable range. Whenever a value is inserted or updated, the database server verifies that the value conforms to the rule before it performs the operation.

You should implement data validation in your Enterprise Objects Framework application whether or not you use database rules. Depending on the nature of the validation, use a formatter or implement an appropriate **validate...** method in your enterprise object class. For more information, see the chapter "Designing Enterprise Objects" on page 65.

Constraints for Enforcing Relational Integrity Rules

Many databases provide mechanisms to enforce relational integrity rules. For example, you can define a constraint (or trigger) that prevents the deletion of a Department that still contains Employees. Enterprise Objects Framework also provides mechanisms for enforcing these types of rules. For example, you can specify delete rules for relationships in EOModeler.

If you use database triggers and constraints, you will have to duplicate the logic in your Enterprise Objects Framework application. In some cases, the duplication won't hurt anything, but in other cases you have to provide special handling to avoid run-time errors.

For example, suppose you have a constraint specifying that you can't delete a department if it still has employees. In addition, you specify the Deny delete rule on the Department entity's **employees** relationship. When a user attempts to delete a department, Enterprise Objects Framework verifies that the corresponding Department object has no employees. If the department has one or more employees, the Framework doesn't allow the delete.

Further suppose that a user moves all the employees from one department to another, deletes the empty department, then saves all changes. Enterprise Objects Framework analyzes the object graph to determine what operations have taken place. It orders the operations by analyzing the relationships and identifying "master" and "detail" entities. In this example, the Department object, the master, would not be deleted until all the employees are updated to reflect their new department. In most cases, Enterprise Objects Framework just does the "right thing." However, if you discover a sequencing problem with your application, you can customize the order in which database operations are performed. For a complete description of the Framework's default ordering algorithm and how to programmatically reorder operations, see the section "How Do I Order Database Operations?" on page 248.

How Do I Invoke a Stored Procedure?

To invoke a stored procedure from your Enterprise Objects Framework application, you must define the stored procedure in a model and decide how to invoke it.

If your stored procedure is defined in the database at the time you create your model, you don't have to do anything to define it. When you create a new model with EOModeler, the application reads stored procedure definitions from the database's data dictionary and stores them in the model's **.eomodeld** file. However, you can also add a stored procedure definition to an existing model. For more information, see the book *Enterprise Objects Framework Tools and Techniques*.

Depending on what a stored procedure does, you can either invoke it explicitly or specify that the Framework invoke it for common database operations.

Invoking a Stored Procedure Automatically

You can define stored procedures to perform the following operations:

Operation	Description
FetchAllProcedureOperation	Fetches all the objects for an entity.
FetchWithPrimaryKeyProcedureOperation	Fetches an object by its primary key.
InsertProcedureOperation	Inserts a new object.
DeleteProcedureOperation	Deletes an object.
NextPrimaryKeyProcedureOperation	Generates a new primary key value.

By associating a stored procedure with an entity's operation, the Framework invokes it automatically when the operation occurs. For example, if you want to use a stored procedure to insert new Customer objects:

1. Define the stored procedure in the database.
2. Define the stored procedure in the model.

3. Associate the stored procedure with the Customer entity's insert operation.

You can associate a stored procedure with an entity using EOModeler as described in the book *Enterprise Objects Framework Tools and Techniques*. Or you can do it programmatically using EOEntity's **setStoredProcedure** method (**setStoredProcedure:forOperation:** in Objective-C). For more information on the operations and on setting them programmatically, see the EOEntity class specification in the *Enterprise Objects Framework Reference*.

Requirements for Framework-Invoked Stored Procedures

When Enterprise Objects Framework invokes a stored procedure for an operation, the procedure must behave in an expected way. The Framework specifies what a stored procedure's arguments, results, and return values should be. The following sections summarize the requirements for each operation.

FetchAllProcedureOperation

The FetchAllProcedureOperation (EOFetchAllProcedureOperation in Objective-C) fetches all the objects for a particular entity. A stored procedure for this operation should have no arguments and return a result set (or in the case of Oracle, a REFCURSOR argument) for all the objects in the corresponding entity.

The rows in the result set must contain values for all the columns Enterprise Objects Framework would fetch if it were not using the stored procedure, and it must return them in the same order. In other words, the stored procedure should return values for primary keys, foreign keys used in class property joins, class properties, and attributes used for locking (generally, values for all the entity's attributes). Also, the stored procedure should return the values in alphabetical order based on the names of their corresponding EOAttribute objects. For example, a Studio entity has the attributes **studioId**, **name**, and **budget**. A stored procedure that fetches all the Studio objects should return the value for a studio's budget value, then the studio's name, and then its studioId.

If an FetchAllProcedureOperation stored procedure has a return value, Enterprise Objects Framework ignores it.

FetchWithPrimaryKeyProcedureOperation

The FetchWithPrimaryKeyProcedureOperation (EOFetchWithPrimaryKeyProcedureOperation in Objective-C) fetches a single enterprise object by its primary key value. A stored procedure for this operation should take an "in" argument for each of the entity's primary key attributes. The argument names must match the names of the primary key attributes. For example, a Studio entity has one primary key attribute named "studioId". As defined in a model, the stored procedure's argument must also be named "studioId".

An FetchWithPrimaryKeyProcedureOperation stored procedure should return a result set (or in the case of Oracle, a REFCURSOR argument) containing the matching row. The row must be in the same form as those returned by an FetchAllProcedureOperation stored procedure.

If an FetchWithPrimaryKeyProcedureOperation stored procedure has a return value, Enterprise Objects Framework ignores it.

InsertProcedureOperation

The InsertProcedureOperation (EOInsertProcedureOperation in Objective-C) inserts a new enterprise object. A stored procedure for this operation should take "in" arguments for each of the corresponding entity's attributes. The argument names must match the names of the corresponding EOAttribute objects.

An InsertProcedureOperation stored procedure should not return a result set. Also, if an InsertProcedureOperation stored procedure has a return value, Enterprise Objects Framework ignores it.

DeleteProcedureOperation

The DeleteProcedureOperation (EODeleteProcedureOperation in Objective-C) deletes a single enterprise object by its primary key value. A stored procedure for this operation should take an "in" argument for each of the entity's primary key attributes. The argument names must match the names of the primary key attributes as in FetchWithPrimaryKeyProcedureOperation stored procedures.

An DeleteProcedureOperation stored procedure should not return a result set. Also, if an DeleteProcedureOperation stored procedure has a return value, Enterprise Objects Framework ignores it.

NextPrimaryKeyProcedureOperation

The NextPrimaryKeyProcedureOperation (EONextPrimaryKeyProcedureOperation in Objective-C) generates a unique primary key value for a new enterprise object. A stored procedure for this operation should take an "out" argument for each of the entity's primary key attributes. The argument names must match the names of the primary key attributes as in FetchWithPrimaryKeyProcedureOperation stored procedures.

An NextPrimaryKeyProcedureOperation stored procedure should not return a result set. Also, if an NextPrimaryKeyProcedureOperation stored procedure has a return value, Enterprise Objects Framework ignores it.

Invoking a Stored Procedure Explicitly

Some stored procedures can't be associated with a specific database operation that Enterprise Objects Framework invokes. For example, if you've defined a stored procedure to return the sum of revenues for all the Movie objects, you'll have to invoke it explicitly. To invoke a stored procedure explicitly, you use an EOAdaptorChannel object. The following code excerpt shows how to do it:

In Java:

```
EOAdaptorChannel adChannel;      // Assume this exists.
EOStoredProcedure sumOfRevenue;
NSDictionary results;
EOModelGroup defaultGroup = EOModelGroup.defaultGroup();

sumOfRevenue = defaultGroup.storedProcedureNamed(
  "sumOfRevenue");
adChannel.executeStoredProcedure(sumOfRevenue, null);
results = adChannel.
  returnValuesForLastStoredProcedureInvocation();
```

In Objective-C:

```
EOAdaptorChannel *adChannel;      // Assume this exists.
EOStoredProcedure *sumOfRevenue;
NSDictionary *results;

sumOfRevenue = [[EOModelGroup defaultGroup]
     storedProcedureNamed:@"sumOfRevenue"];
[adChannel executeStoredProcedure:sumOfRevenue
withValues:nil];
results =
     [adChannel
returnValuesForLastStoredProcedureInvocation];
```

The method **returnValuesForLastStoredProcedureInvocation** returns stored procedure parameter and return values. The dictionary returned by this method (**results** in this example) has entries whose keys are the names of the stored procedure's out and in-out arguments. The dictionary may also contain an entry with the key "returnValue" whose value is the return value of a stored procedure (if it has one).

Tip: If you're using Sybase, the return values dictionary always contains a "SybaseStoredProcedureReturnStatus" key whose value is the return status of the stored procedure. You don't need to declare an output parameter for this.

Tip: If you're using Oracle, you can define a stored procedure to represent a function. Add an argument named "returnValue" and use the EOAdaptorChannel method **returnValuesForLastStoredProcedureInvocation** to get the function's result.

If you want to invoke a stored procedure that returns rows, you use **fetchRow** (**fetchRowWithZone:** in Objective-C) as you would if you were fetching the results of a **selectAttributes** message (**selectAttributes:fetchSpecification:lock:entity:** in Objective-C). For example, the following code excerpts fetch Movie objects using the **fetchMovies** stored procedure:

In Java:

```
EOAdaptorChannel adChannel;      // Assume this exists.
EOStoredProcedure fetchMovies;
NSDictionary row;
EOModelGroup defaultGroup = EOModelGroup.defaultGroup();

fetchMovies =
defaultGroup.storedProcedureNamed("fetchMovies");
adChannel.executeStoredProcedure(fetchMovies, null);

while (adChannel.isFetchInProgress()) {
  while (row = adChannel.fetchRow()) {
      // Process theRow.
  }
}
```

In Objective-C:

```
EOAdaptorChannel *adChannel;      // Assume this exists.
EOStoredProcedure *fetchMovies;
NSDictionary *row;

fetchMovies = [[EOModelGroup defaultGroup]
     storedProcedureNamed:@"fetchMovies"];
[adChannel executeStoredProcedure:fetchMovies
withValues:nil];

while ([adChannel isFetchInProgress]) {
  while (row = [adChannel fetchRowWithZone:nil]) {
      /* Process theRow. */
  }
}
```

Neither of the previous examples uses stored procedures that have arguments. If you want to invoke a stored procedure that does, you provide the argument values to the stored procedure in the **executeStoredProcedure** message (**executeStoredProcedure:withValues:** in Objective-C). For example, the following code excerpts use a stored procedure to insert a row into the database:

In Java:

```
EOAdaptorChannel adChannel;  // Assume this exists.
EOStoredProcedure insert;
NSDictionary row;
EOModelGroup defaultGroup = EOModelGroup.defaultGroup();

// Assume row contains the values for the row to insert

insert = defaultGroup.storedProcedureNamed("insert");
adChannel.executeStoredProcedure(insert, row);
```

In Objective-C:

```
EOAdaptorChannel *adChannel;  // Assume this exists.
EOStoredProcedure *insert;
NSDictionary *row;

// Assume row contains the values for the row to insert.

insert = [[EOModelGroup defaultGroup]
     storedProcedureNamed:@"insertTest"];
[adChannel executeStoredProcedure:insert
withValues:row];
```

Note: The EOAdaptorChannel must be open for this code to work.

For more information on invoking stored procedures explicitly, see the EOAdaptorChannel class specification in the *Enterprise Objects Framework Reference*.

How Do I Order Database Operations?

An Enterprise Objects Framework application typically queues up changes to many enterprise objects before saving the changes to the database. It is then the job of an EODatabaseContext to analyze an object graph to determine what has changed, translate the changes to database operations, and perform the operations using an EOAdaptorChannel.

Enterprise Objects Framework implements a default algorithm for ordering the database operations that reduces the number of scenarios in which you have to reorder adaptor operations programmatically. Enterprise Objects Framework builds an entity ordering by identifying “master” and “detail” entities as follows.

- If an entity (Employee, for example) has a to-one relationship to a second entity (Department) and the inverse relationship is a to-many, then the second entity (Department) is considered the *master*.

- If an entity has a to-one relationship to a second entity and the inverse relationship is also to-one, then the framework checks if one of the relationships propagates its primary key. The source of the "propagatesPrimaryKey" relationship is considered to be the master entity.

Before sending operations to the database, Enterprise Objects Framework orders the operations based on these master definitions. The operations will have the following order:

1. Lock operations (master entities before detail entities)

2. Inserts (master entities before detail entities)

3. Updates (master entities before detail entities)

4. Deletes (detail entities before master entities)

However, if your database uses sophisticated referential integrity, if it uses triggers, or there are referential integrity constraints that are not modeled in EORelationships, you may still need to reorder adaptor operations programmatically.

For example, if Employees have to-one relationships to their managers, then you will have to explicitly order the database operations such that a manager is inserted before that manager's direct reports are inserted. Enterprise Objects Framework can't catch this case because the relationship is self-referential.

Another example of when you might reorder database operations is when you want to use the same ordering algorithm that other non-Enterprise Objects Framework applications are using to prevent deadlock contention problems (such as can occur with Sybase servers). If a Framework application takes locks in a different order than other non-Framework applications, then you might encounter deadlock problems.

You can order database operations by implementing either or both of the following EODatabaseContext delegate methods

In Java:

- databaseContextWillOrderAdaptorOperations
- databaseContextWillPerformAdaptorOperations

In Objective-C:

- databaseContext:willOrderAdaptorOperationsFromDatabaseOperations:
- databaseContext:willPerformAdaptorOperations:adaptorChannel:

The "willOrder" method provides the delegate with more information from the object graph than the "willPerform" method. However, "willPerform" can be more convenient. Its second argument is an array of adaptor operations that are already prepared. The delegate only needs to rearrange them. For more information on these delegate methods, see the EODatabaseContext class specification in the *Enterprise Objects Framework Reference*.

How Are Enterprise Objects Cleaned Up?

If you use an EODisplayGroup to fetch enterprise objects into your application, you might wonder:

- Who "owns" the objects?
- How do they get cleaned up?
- How are their snapshots cleaned up?
- What happens if you have reference cycles?

In applications that fetch a lot of enterprise objects or are long-running, these are important questions. How they're answered depends on what language you're using. However, regardless of language, you don't typically have to worry about any of these issues. With Java's garbage collection, enterprise objects and their related resources are automatically cleaned up when they are no longer in use.

In Objective-C, as long as you follow the object ownership conventions defined in the Foundation framework, enterprise objects and their related resources are similarly deallocated automatically. The following sections provide more information on how this happens in Objective-C Enterprise Objects Framework applications.

For more general information on this automatic object disposal mechanism in Objective-C, see the introduction to the *Foundation Framework Reference*.

Who Owns an Enterprise Object?

In design terms, one object might own another; but in the Foundation Framework, no object really "owns" another. Rather, one or more objects may "retain" another object. If one object retains another, it has a responsibility to release it when it no longer needs the other object. In Enterprise Objects Framework applications, an enterprise object is retained by other enterprise objects that have a relationship to it. An enterprise object is also retained by an EODisplayGroup object that fetches and displays it.

How Does an Enterprise Object Get Deallocated?

In a Enterprise Objects Framework applications, an enterprise object is retained by other enterprise objects that have a relationship to it and by any EODisplayGroup objects that fetch and display it. Typically, enterprise objects are deallocated automatically when they are no longer referenced by other objects. You don't ordinarily manage the deallocation of enterprise objects explicitly.

Accessor methods that manage relationships to one or more enterprise objects also release objects when they no longer need to reference them. For example, the following method releases an employee's old manager before assigning a new one:

```
- (void)setManager:(Employee *)aManager
{
  [self willChange];
  [manager autorelease];
  manager = [aManager retain];
}
```

If an enterprise object class doesn't implement accessor methods for a relationship, the Framework automatically releases and retains the destination objects. Similarly, an EODisplayGroup object releases its enterprise objects immediately before it fetches a new set of objects or immediately before it is deallocated itself. Unless you explicitly retain an enterprise object, it is automatically deallocated when its display group stops displaying it.

If you *do* explicitly retain an enterprise object (either by sending it a retain message or by adding it to a collection), the enterprise object is not deallocated until you release it (either by sending it a **release** message or, if it's in a collection, by releasing its collection).

Methods for getting enterprise objects without using an EODisplayGroup don't automatically retain objects. For example, the objects returned from EODataSource's **fetchObjects** method and EOEditingContext's **objectsWithFetchSpecification:** method are not retained by any object. Unless you retain them, they will be deallocated automatically.

How Are an Object's Snapshots Deallocated?

Enterprise Objects Framework keeps two kinds of snapshots:

- Object snapshots that are maintained by EOEditingContexts
- Row snapshots that are maintained by EODatabaseContexts

An object snapshot is deallocated at the same time its enterprise object is deallocated. A row snapshot, however, is only invalidated when its EODatabaseContext is deallocated or when it receives an **invalidateAllObjects** message or **invalidateObjectWithGlobalID:** message.

Multiple EOEditingContexts may use a single EODatabaseContext object and its row snapshots. As a result, it isn't practical to deallocate a row snapshot when a corresponding enterprise object is deallocated. An enterprise object in another EOEditingContext may still reference the snapshot. To deallocate row snapshots explicitly, use one of the **invalidate...** methods.

What Happens If You Have Retain Cycles?

A retain cycle occurs when two objects retain one another. They may retain one another directly, or indirectly through a collection or another object. Retain cycles occur quite commonly in Enterprise Objects Framework applications. For example, if an Employee object has a relationship to a Department object, the Department object probably has a relationship to its employees as well. Normally an object retains the objects to which it has a relationship, so the reciprocal relationships between Employee and Department objects form a retain cycle.

Objects in a cycle stay in memory until the cycle is broken. If the cycle is never broken, the objects stay in memory until the process exits. Too many unbroken retain cycles degrade an application's performance.

One strategy for handling retain cycles is to ensure that none are created. If you don't need reciprocal relationships, don't create them. Reciprocal relationships, however, are very useful. You are more likely to use one of the following approaches for handling retain cycles.

invalidateObjectsWhenFreed

Retain cycles between objects can be broken automatically when their EOEditingContext is deallocated. To break retain cycles automatically, set the EOEditingContext's **invalidatesObjectsWhenFreed** attribute to YES, which is the default. This approach works well in multi-document applications in which EOEditingContexts are deallocated when their windows close.

invalidateAllObjects

In applications that aren't multi-document, you can break cycles by sending an **invalidateAllObjects** message to an EOEditingContext's root EOObjectStore. You typically invalidate enterprise objects after saving changes to the database or after reverting.

This method replaces all the associated enterprise objects with EOFault objects, eliminating retain cycles in the process. It has the side-effect of invalidating all the enterprise objects in a peer editing context as well.

Should I Make Foreign Key Attributes Class Properties?

You should *not* make foreign key attributes class properties. If you need to access a foreign key value (because you want to display it in the user interface, for example), you should access it through the corresponding destination object.

Class properties that are foreign keys can become out of sync with their corresponding destination objects. For example, assume that an Employee class defines a relationship, **department**, to its department and has a class property, **departmentID**, for the corresponding foreign key. Assigning an employee to a new department doesn't update the **departmentID** property in the employee object until the enterprise object is saved to the database. Thus, **departmentID** contains the primary key value for the old department while the **department** relationship points to the new department.

Instead of making the foreign key a class property of an enterprise object, you should implement a method that gets the value from the destination object. For example:

In Java:

```
public Object departmentID() {
  NSDictionary primaryKey =
      EOUtilities.primaryKeyForObject(
          department.editingContext(),
          department);
  return primaryKey.objectForKey("departmentID");
}
```

In Objective-C:

```
- (id)departmentID
{
  NSDictionary *primaryKey = [[department editingContext]
      primaryKeyForObject:department];
  return [primaryKey objectForKey:@"departmentID"];
}
```

In the Java implementation, the EOUtilities static method **primaryKeyForObject** returns the primary key dictionary for the **department** object. In Objective-C, the method is **primaryKeyForObject:**, which is added to the EOEditingContext description through the EOUtilities category in EOAccess.

How Do I Share Models Across Applications?

You should put shared models in a shared framework. Enterprise Objects Framework automatically looks for models in the frameworks used by your application (both at run-time, and at design time in EOModeler, Interface Builder, and WebObjects Builder). Also put the enterprise object classes that correspond to the model in the framework.

In order for Enterprise Objects Framework to find a model in a framework, that framework must be built and installed. During design, Enterprise Objects Framework looks at the model in the installed version of the framework (not in the source version of the framework project). This can result in Interface Builder and WebObjects Builder not seeing the changes in the source version of the model since it's looking at the version in the installed framework, rather than at the one in your source directory. You can tell Enterprise Objects Framework to look for models in the source version of your framework projects by using the following commands (executed in a shell):

```
defaults write NSGlobalDomain
EOProjectSourceSearchPath"($(HOME)/myProjectsDirectory1,
/myOtherProjectsDirectory)"
```

Then, when EOModeler, Interface Builder, or WebObjects Builder look for models contained in one of your frameworks, it first searches all project directories within **$(HOME)/myProjectsDirectory1** and **/myOtherProjectsDirectory** before searching for the built versions.

Appendix

Entity-Relationship Modeling

A database server stores data in the structures that it defines: A relational database uses tables to store data, an object-oriented database uses objects, a file system uses files, and so on. The Enterprise Objects Framework uses the terminology of *Entity-Relationship modeling* (or *E-R modeling*) to describe a server's data structures in a way that allows those data structures to be mapped to enterprise objects.

Entity-Relationship modeling isn't unique to the Enterprise Objects Framework; it's a popular discipline with a set of rules and terms that are documented in database literature. The Enterprise Objects Framework uses a modified version of the traditional rules of E-R modeling.

When your data store is a relational database, you can use the EOModeler application to specify the mapping between the database data and your enterprise objects. The model file you produce using EOModeler describes the server's data structures in terms that the Enterprise Objects Framework can understand. Note that if you're working with a data store other than a database, you must create your own data structures to map the server's data to your enterprise objects.

This chapter presents the E-R terms and concepts as they are used by the Framework. For instructions on putting these concepts into practice, see the book *Enterprise Objects Framework Tools and Techniques*.

Modeling Objects

In an Entity-Relationship model, distinguishable things are known as *entities*, each entity is defined by its component *attributes*, and the affiliations, or *relationships*, between entities are identified (together, attributes and relationships are known as *properties*). From these three simple *modeling objects*, arbitrarily complex systems can be modeled. For instance, a company's customer base, a library of books, or a network of computers can all be depicted as E-R models. If the parts of a system can be identified, the system can be expressed as an E-R model.

Pure Entity-Relationship modeling is independent of native database architecture. Theoretically, an E-R model can be implemented as a relational database, an object-oriented database, a file system, or any other data storage system. In practice, E-R modeling fits most naturally with relational databases; in other words, with databases that store data in two-dimensional tables. The examples and illustrations in this chapter follow this lead by posing a hypothetical relational database server from which data is drawn.

Entities and Attributes

Entities and attributes represent structures that contain data. In a relational database, entities represent tables; an entity's attributes represent the table's columns. A sample table that could be represented by an **Employee** entity is shown below:

EMPLOYEE		
LAST_NAME	FIRST_NAME	PHONE
Winton	James	415-023-7265
Veasey	Kai	415-023-2611
MacAskill	Jane	415-028-4407
Maselli	John	
Lunau	Ken	415-322-2815
Windgate	Mark	415-021-3718
DeKeyser	John	415-012-3456
Kanzaki	Mark	415-022-3100
Kallimani	Yoshinori	
Fisk	George	415-026-4605
Davidson	Cary	415-033-3333

Figure 53. The "EMPLOYEE" Table

Each row in the table can be thought of as an "instance of an entity." Thus, an employee record is called an instance of the **Employee** entity. In the Enterprise Objects Framework, each instance of an entity typically maps to one enterprise object.

Contained within an entity is a list of features, or attributes, of the thing that's being modeled. The **Employee** entity would contain attributes such as the employee's last name, first name, phone number, and so on. This simple model is depicted in Figure 54.

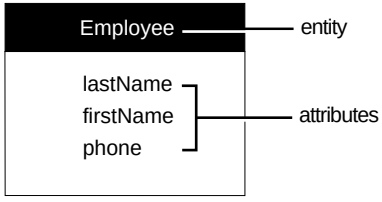


Figure 54. The **Employee** Entity

In traditional E-R modeling, each entity represents all or part of one database table. The Enterprise Objects Framework allows you to go beyond this, however, by adding attributes to an entity that actually reflect data in other, related tables (the process of adding attributes from other entities is known as *flattening*). An entity in the Framework is analogous to a database view; in a sense it's a virtual table that maps to one or more real database tables.

Entities can also have *derived* attributes, which do not correspond directly to any of the columns in a database table. Frequently, these are computed from one or more attributes. For instance, a derived attribute could be used to automatically compute an employee's annual salary by multiplying his monthly salary (obtained from a simple monthly salary attribute) by twelve.

Enterprise objects are based on entities. Typically, each of an entity's properties are represented in the enterprise object as instance variables (although this is not a requirement). Enterprise objects can have instance variables that do not correspond to any of the entity's properties.

Names and the Data Dictionary

The table and column names shown in Figure 54 are the names that a hypothetical server might use. The collection of a server's table and column names is called its *data dictionary*. In your application, you can't refer directly to items in the server's data dictionary. To identify the server's "EMPLOYEE" table, for example, you must refer to the entity that represents the table—in other words, the **Employee** entity. The correspondence between the server's names and the names of the modeling objects that you create isn't coincidental; you have to tell each modeling object which data dictionary name it represents. This is done as you create the model.

Server names (in other words, names in a server's data dictionary) can be case-insensitive (depending on the database server). The names of modeling objects, on the other hand, are always case-sensitive. Throughout this chapter (and the rest of this manual) modeling objects are given names that match, except for case, the corresponding dictionary names (given the hypothetical relational database server that's used in the examples). To further distinguish the two, server names are uppercase and quoted—for example, the "EMPLOYEE" table—while modeling object names use a different font: **AnEntity**, **anAttribute**, **aRelationship**. Note that entity names are capitalized like class names, while attribute and relationship names are lowercase with intervening capital letters. Attributes are occasionally identified by their *definition*, with the entity and attribute names connected by a period: **AnEntity.anAttribute**.

Attribute Data

When you use an attribute to identify a particular datum in a table, you refer to the value *for* that attribute, given a particular record. An employee's phone number, for example, is the value *for* the **Employee.phone** attribute. The "value for an attribute" construction enforces the notion that the attribute itself doesn't contain data.

Not every employee will necessarily have a phone number. If a record's value for a particular attribute can't be determined (or doesn't exist), the value is said to be NULL.

Data Types

Every database attribute is assigned a data type (such as **int**, **String**, and so on). All values for a particular attribute take the data type of that attribute. In other words, the values in a particular column are all of the same type. When an enterprise object is fetched from the database, the value for each attribute is converted from its external data type into a suitable scalar or value class type that can be used by the enterprise object. For example, a Sybase varchar would become a java.lang.String (or NSString in Objective-C) in an enterprise object.

None of the candidate data types allow lists of data; the value for a particular attribute in a particular record must be a single datum. Thus, in addition to indicating that an employee has a last name, a first name, and a phone number, the diagram in Figure 54 indicates that every employee has a *single* last name, a *single* first name, and a *single* phone number (where any of these single values can be NULL). This "atomic attribute rule" will become particularly important in the discussion of relationships, later in this chapter.

Attribute Types

An attribute may be *simple*, *derived*, or *flattened*. A simple attribute corresponds to a single column in the database, and may be read or updated directly from or to the database.

A derived attribute doesn't correspond to a single database column and is usually based on some other attribute, which is modified in some way. For example, if an **Employee** entity has a simple monthly salary attribute, you could define a derived **annualSalary** attribute as "salary * 12". Derived attributes, since they don't correspond to real values in the database, are effectively read-only; it makes no sense to write a derived value.

A flattened attribute (which, in the Enterprise Objects Framework, is a special type of derived attribute) is actually an attribute of some other entity reached through a *relationship*. A flattened attribute's *definition* consists of one or more relationships separated by periods, ending in an attribute name. For example, if the **Employee** entity has the relationship **toDepartment** and the **Department** entity has the attribute **departmentName**, you can define **employeeDeptName** as an attribute of your **Employee** entity by creating an attribute for it with a definition of "**toDepartment.departmentName**".

The Primary Key

Each of the records in a table must be unique—no two records can contain exactly the same values. To ensure this, each entity must contain an attribute that's guaranteed to represent a unique value for each record. This attribute is called the entity's *primary key*.

The **Employee** entity, as defined above, doesn't contain a primary key. If the company were to hire two employees with the same name, the records for those two employees wouldn't be distinguishable from each other. To amend this, a primary key called **empID**—an attribute for which each distinct employee has a unique value—is added to the **Employee** entity. Figure 55 shows the amended entity; the primary key is marked with a key symbol.

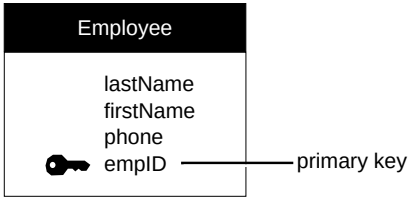


Figure 55. The **Employee** Entity with a Primary Key

The value for a primary key may or may not represent a real-world value. The **empID** attribute used above may, for instance, contain the employee's social security number. Or, it may just contain an arbitrary value used only to distinguish a particular record from other employee records.

An entity can contain any number of attributes that represent unique data, but only one of them needs to be declared as a primary key. Declaring more than one as a primary key creates a *compound primary key*.

Compound Primary Keys

Typically, the primary key for an entity is a single attribute. However, you can designate a combination of attributes as a compound primary key. In a compound primary key, the value for any one of the constituent attributes isn't necessarily unique, but the combination of all of them is.

For example, consider employee time cards. Every time card could be uniquely identified through a combination of its employee number and an additional time card number (to distinguish multiple cards for the same employee). Taken on their own, neither of these numbers is necessarily unique for all time cards, but the combination of the two is. Figure 56 illustrates a **TimeCard** entity in which the attributes **empID** and **timeCardID** form a compound primary key.

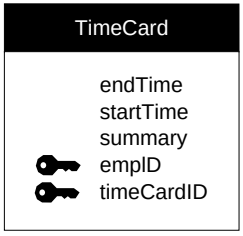


Figure 56. An Entity with a Compound Primary Key

Relationships

Your employee database might have, in addition to the **Employee** entity, a **JobTitle** entity that identifies the various job titles that an employee can have and whether each title represents a salaried or an hourly position. A *relationship* between the **Employee** entity and the **JobTitle** entity expresses the affinity between employees and titles, and allows you to access the title information for a given employee. Graphically, a relationship can be shown as a named arrow that points from one entity (the *source entity*) to another (the *destination entity*); the Employee-JobTitle relationship (which is named **toJobTitle**) is depicted in Figure 57.

Note: To support the **toJobTitle** relationship, the **Employee** entity has been altered—the **titleID** attribute has been added to it. This is explained in the section "Relationship Keys" on page 267.

The table that's represented by the source entity is referred to as the *source table*; the source table contains *source records*. Similarly, the table that's represented by the destination entity is referred to as the *destination table*; it contains *destination records*.

Be aware that you can't just randomly create relationships between your entities. Relationships that you add to your entities must reflect real relationships between the tables in the database. For more information, see the section "Relationship Keys" on page 267.

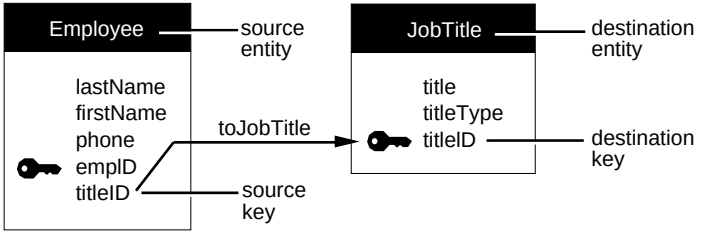


Figure 57. The **toJobTitle** Relationship

Relationship Directionality

Relationships are *unidirectional*. In a unidirectional relationship, the path that leads from the source to the destination can't be traveled in the opposite direction—you can't use a relationship to go from the destination to the source. For example, although you can use the toJobTitle relationship to find the title for a particular employee, you can't use it to get a list of the employees that share a particular title.

Unidirectionality is enforced by the way a relationship is resolved. Specifically, the source record is a given. Resolving a relationship means finding the correct destination record (or records) given a specific source record.

Bidirectional relationships—in which you can look up records in either direction—can be created by adding a separate "return-trip" relationship. This is demonstrated in the section "Bidirectional Relationships" on page 272.

Naming Relationships

Most of the relationships described in this manual use a simple naming convention: relationships are named after the destination entity. For example, a Movie entity can have a **studio** relationship to a Studio entity, and a **roles** relationship to a MovieRoles entity. Note that singular names are typically used for to-one relationships, and plural names are used for

to-many relationships. However, you're not bound by this convention—EOModeler lets you give relationships any names you like.

In the figures throughout this book, the entity that is adjacent to the relationship's label is said to *own* the relationship. For example, in Figure 57 the **Employee** entity owns the **toJobTitle** relationship, as indicated by the proximity of the "toJobTitle" label to the entity.

Relationships and the Data Dictionary

Unlike entities and attributes, relationships don't correspond to names in the server's data dictionary. In general, most servers don't define structural elements for relationships, so their data dictionaries don't contain names to which E-R relationships can correspond. But relationships aren't completely disassociated from the data dictionary: A relationship's definition, as explained in the next section, depends on the existence of particular entities and attributes (which, as described earlier, must correspond to data dictionary names).

Relationship Keys

The construction of a relationship involves more than just two entities. You also have to designate at least one attribute from each entity as a *relationship key*. In the **toJobTitle** relationship, for instance, the **Employee.titleID** and **JobTitle.titleID** are so designated; this is indicated in Figure 57 as the two attributes that lie at either end of the relationship arrow. Just as the tables are called source and destination tables, so are the relationship keys named. In the source entity, the relationship key is called the *source key*. The destination entity's relationship key is called the *destination key*.

Note: As in the case of the **toJobTitle** relationship, the source and destination keys often have the same name, although this isn't a requirement of model design.

The reason you need to designate relationship keys is so the relationship can be used to create cross-references between specific instances of the related entities (this is called "resolving" the relationship). For example, let's say you fetch an employee object. The Enterprise Objects Framework takes the value for the employee's **titleID** attribute and

compares it to the value for **titleID** in each **JobTitle** instance. A match locates the desired job title record.

For this cross-referencing scheme to work, the source and destination keys must characterize the same data—you couldn't find an employee's job title by comparing, for example, **Employee.empID** to **JobTitle.titleID**. This is why the **titleID** attribute was added to the **Employee** entity.

An Example with Data

To further illustrate how a relationship is resolved, consider the "EMPLOYEE" and "JOB_TITLE" tables presented in Figure 58 (for the purpose of this example, only the essential columns are shown).

Here we see that the value for the **titleID** attribute for James Winton is 1. Looking in the "JOB_TITLE" table, we see that 1 is the ID of the President. Thus, James Winton is the company president. Similarly, we can determine that Kai Veasey is a manager.

EMPLOYEE		
LAST_NAME	FIRST_NAME	TITLE_ID
Winton	James	1
Veasey	Kai	3
MacAskill	Jane	7
Maselli	John	5
Lunau	Ken	4
Windgate	Mark	4
DeKeyser	John	7
Kanzaki	Mark	6
Kallimani	Yoshinori	4
Fisk	George	6
Davidson	Cary	4

JOB_TITLE	
TITLE	TITLE_ID
President	1
Vice President	2
Manager	3
Engineer	4
Designer	5
Sales Representative	6
Administrator	7

Figure 58. The "EMPLOYEE" and "JOB_TITLE" Tables

Choosing Relationship Keys

Any attribute can be used as a relationship key, but some are better suited than others. In general, of the two relationship keys for a particular relationship, the destination key will be a primary key for its entity (or, otherwise, an attribute that characterizes unique data) and the source key is manufactured to emulate the destination key. In traditional E-R modeling, the emulating attribute is called a *foreign key*. The **toJobTitle** relationship demonstrates this: The destination key in the **JobTitle** entity is **titleID**, the primary key for that entity. The **titleID** attribute is added to **Employee** as foreign key.

Note that if **empID** had been used as the relationship key for the **toJobTitle** relationship, a given title could only be assigned to a single employee.

Compound Relationship Keys

A relationship's keys needn't be single attributes from the related entities; any number of attributes can be paired as relationship keys within the same relationship to form a *compound relationship key.* A relationship that designates more than one pair of keys is called a *compound relationship*.

For example, consider an entity (**empPhoto**) containing the employee's picture that uses the attributes **firstName** and **lastName** as a compound relationship key. (Using people's names for unique identification is generally a bad idea, but it serves the purpose for illustration. In actual practice, this relationship would likely use **empID** as its relationship key.) This relationship is depicted in Figure 59.

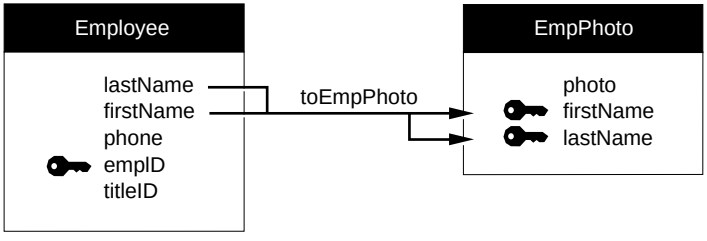


Figure 59. A Compound Relationship

The algorithm used to resolve a compound relationship is similar to that for a simple relationship. The only difference is the number of pairs of relationship key values that are compared. For two records to correspond, all of the comparisons must be successful.

Note: The keys in a compound relationship can be a combination of *any* attributes—not just a compound primary key (or foreign keys to a compound primary key). Conversely, you can use a single attribute from a compound primary key as a relationship key in a simple (non-compound) relationship.

Joins

Relationships are made up of source-destination key pairs. A *join* is the pairing of one source attribute and one destination attribute for purposes of establishing a relationship. Thus, simple relationships consist of one join. Compound relationships are composed of two or more joins. In Figure 59, for example, the **toEmpPhoto** relationship is composed of two joins: one linking **Employee.lastName** to **EmpPhoto.lastName**, and one linking **Employee.firstName** to **EmpPhoto.firstName**.

The Enterprise Objects Framework requires you to declare the join in a relationship as either an *inner join*, a *right outer join*, a *left outer join*, or a *full outer join*. These four *join semantics* are defined as follows:

- In an inner join, if a destination record can't be found for a given source record, that source record isn't included in the result of the join. Destination records that don't match up to any records in the source table are not included in the result of an inner join, either.
- In a right outer join, destination records for which no source record can be found are included, but not the reverse.
- In a left outer join, source records for which no destination record can be found are included, but not the reverse.
- In a full outer join, *all* source records from both tables are included in the result of the join.

Relationship Cardinality

Every relationship has a *cardinality*; the cardinality tells you how many destination records can (potentially) resolve the relationship. The Enterprise Objects Framework defines two cardinalities, *to-one* and *to-many*:

- In a to-one relationship, for each source record there's *exactly one* corresponding destination record.
- In a to-many relationship, for each source record there may be zero, one, or more corresponding destination records.

The **toJobTitle** relationship is an example of a to-one relationship: An employee can only have one title. The converse relationship, from **JobTitle** to **Employee**, would be to-many: a single title can be shared by more than one employee, or there may be no employees with a given title. This relationship, which is owned by **JobTitle** and called **toEmployee**, is shown in Figure 60 (for clarity, the source and destination components are pointed out). That the relationship is to-many is indicated by the double arrowhead.

Notice that the relationship keys for the **toEmployee** relationship are the same as for **toJobTitle**. However, the source and destination key assignments are reversed. In other words, whereas **Employee.titleID** is the source key for the **toJobTitle** relationship, it's the destination key for **toEmployee**; similarly, **JobTitle.titleID** changes destination and source key roles between the two relationships.

This switch does more than demonstrate that the same attributes can be used as relationship keys in more than one relationship; it also exemplifies the typical orientation of the primary key with regard to the relationship keys in to-one and to-many relationships:

- In a to-one relationship, the destination key is always the primary key for its entity.
- In a to-many relationship, the source key is usually a primary key.

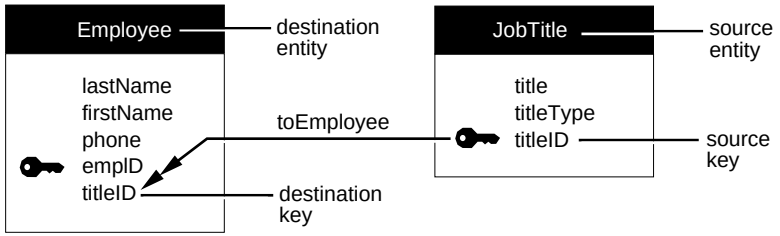


Figure 60. A To-Many Relationship

Bidirectional Relationships

Since relationships, as defined by the Enterprise Objects Framework, are unidirectional, it's natural to assume that to simulate a bidirectional relationship—in other words, to express the natural relationship between two entities without regard for direction—all you need is two relationships: One that leads from entity A to entity B, and one that leads from entity B to entity A. Unfortunately, it isn't always that easy.

Consider, for example, the actual relationship between employees and projects. A project can involve many employees, and a single employee can contribute to more than one project.

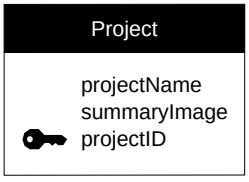


Figure 61. The **Project** Entity

Forming a to-many relationship between **Employee** and **Project** (**toProject**) and a to-many relationship between **Project** and **Employee** (**toEmployee**) doesn't work, because it's impossible to assign relationship keys that would support this set-up. For example, in the **toProject** relationship you can't use the **empID** attribute as a source key because the destination key, **Project.empID** (added as a foreign key), wouldn't be atomic (since a project may consist of more than one employee). Importing **projectID** as a foreign key into **Employee** has the same problem: The attribute wouldn't be atomic (since an employee may be involved with more than one project).

The most common way to establish this "many-to-many" relationship (as it's called in traditional E-R modeling) is to insert an auxiliary entity between **Employee** and **Project**, and form a network of relationships to and from it. This is depicted in Figure 62.

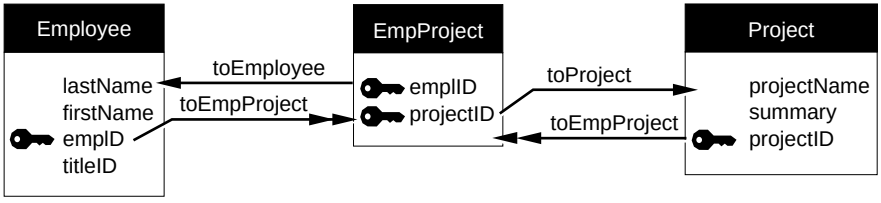


Figure 62. A Many-to-Many Relationship

The compound primary key used in **EmpProject** indicates that the entity characterizes unique combinations of employees and projects. The table that the entity represents would hold a different record for each employee of every project. For example, if three employees were involved with a single project, there would be three **EmpProject** instances with the same value for the **projectID** attribute, but each record would have a different value for its **empID** attribute.

The Tables Behind the Many-to-Many Model

To better understand how the many-to-many model works, it helps to see an example of the tables that store the data. Sample "EMPLOYEE" and "PROJECT" tables that are filled with this information are shown in Figure 63 (for clarity, only relevant attributes are shown).

EMPLOYEE		
LAST_NAME	FIRST_NAME	EMP_ID
Winton	James	101
Veasey	Kai	102
MacAskill	Jane	103
Maselli	John	106
Lunau	Ken	107
DeKeyser	John	108
Windgate	Mark	109
Kanzaki	Mark	112
Kallimani	Yoshinori	116
Fisk	George	134
Davidson	Cary	137

PROJECT	
PROJECT_NAME	PROJECT_ID
Info Environment	503
Net DesignWorks	507
Info Vault	510

Figure 63. "EMPLOYEE" and "PROJECT" Tables

The "EMP_PROJECT" table is shown in Figure 64 (for clarity, the last names and project names are shown in the margins).

	EMP_PROJECT		
	EMP_ID	PROJECT_ID	
Winton	101	507	Net DesignWorks
Veasey	102	503	Info Environment
Veasey	102	510	Info Vault
MacAskill	103	507	Net DesignWorks
Maselli	106	507	Net DesignWorks
Maselli	106	503	Info Environment
Lunau	107	510	Info Vault
Windgate	109	510	Info Vault
DeKeyser	108	503	Info Environment
Kanzaki	112	507	Net DesignWorks
Kallimani	116	503	Info Environment
Kallimani	116	510	Info Vault
Fisk	134	507	Net Designworks
Davidson	137	510	Info Vault

Figure 64. The "EMP_PROJECT" Table

As expected, some values appear more than once for the **empID** attribute; similarly, some values for **projectID** are repeated. But since **empID** and **projectID** form a compound primary key for the **EmpProject** entity, no two records may possess the same combination of values for these two attributes. This fact—that no two records can have the same empID and the same ProjectID—signifies that a given employee cannot be assigned to a single project more than once.

Reflexive Relationships

The source and destination entities in a relationship needn't be different. Where the entities in a relationship are the same, a *reflexive relationship* is created. Reflexive relationships are important in characterizing a system in which an instance of an entity points to another instance of the same entity.

For example, to show who a given employee reports to, you could create a separate **Manager** entity. It would be easier, however, to just create a reflexive relationship, as shown in Figure 65.

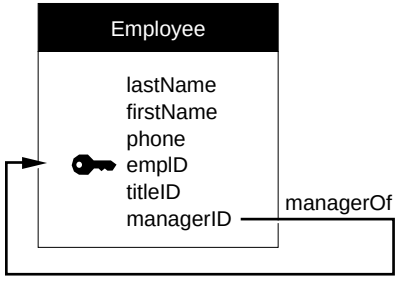


Figure 65. A Reflexive Relationship

Note: The name of the relationship, **managerOf**, doesn't follow the relationship naming convention suggested earlier in this chapter. However, it follows from the meaning of the relationship, and meaning takes precedence over form.

The **managerID** attribute acts as the relationship's source key; **empID** is the destination key. Where an employee's **managerID** matches another employee's **empID**, the first employee reports to the second. If an employee doesn't have a manager, the value for the **managerID** attribute is NULL in that employee's record.

Reflexive relationships can represent arbitrarily deep recursions. Thus, from the model above, an employee can report to another employee who reports to yet another employee, and so on. This could go on until an employee who's **managerID** is NULL is reached, denoting an employee who reports to no one (probably the company president!).

Flattened Attributes

At the beginning of this chapter, it was stated that an entity maps to a table in the database. This is not strictly true, however, because the Enterprise Objects Framework allows you to add *flattened attributes* (and *flattened relationships*) to your entity, effectively extending the entity's mapping to more than one table in a database.

A flattened attribute is an attribute that you effectively add from one entity to another by traversing a relationship. You can't add arbitrary attributes from various entities, however. To add an attribute from one entity to another, there must be a to-one relationship between those entities.

For example, by traversing the **toJobTitle** relationship, you can determine a given employee's title. If you add the **title** attribute from the **JobTitle** entity to the **Employee** entity as a flattened attribute, the Enterprise Objects Framework will automatically traverse the relationship and locate the employee's title when the employee is fetched from the database.

To your code, the flattened attribute looks like any other. After adding the **title** attribute to the **Employee** entity as a flattened attribute (which has no effect on the "EMPLOYEE" table in the database), for instance, your application's view of the **Employee** table would look like Figure 66:

EMPLOYEE		
LAST_NAME	FIRST_NAME	JOB_TITLE
Winton	James	President
Veasey	Kai	Manager
MacAskill	Jane	Administrator
Maselli	John	Designer
Lunau	Ken	Engineer
Windgate	Mark	Engineer
DeKeyser	John	Administrator
Kanzaki	Mark	Sales Representative
Kallimani	Yoshinori	Engineer
Fisk	George	Sales Representative
Davidson	Cary	Engineer

Figure 66. A View of the "EMPLOYEE" Table After Adding a Flattened Attribute

You are not limited to flattening attributes across a single relationship; any number of relationship traversals can be employed. Thus, if there was a relationship between the **JobTitle** entity and a **SalaryRange** entity, you could include an employee's maximum salary with the rest of the employee information by flattening a **toJobTitle.toSalaryRange.maxSalary** attribute into the **Employee** entity.

Flattened Relationships

Just as you can flatten an attribute to add it to another entity, so can you flatten a relationship. This gives a source entity access to relationships that a destination entity has with other entities. It is equivalent to performing a multi-table join.

As an example, suppose you need department information for corporate assets that are assigned to employees, using the entities and relationships shown in Figure 67. One way to obtain the needed information is to flatten the relevant attributes (**deptName** and **location**, perhaps) across the **toEmployee** and **toDepartment** relationships. A simpler way would be to flatten the **toDepartment** relationship itself, so that it appears to your code as if the **Department** entity is a part of the **Equipment** entity.

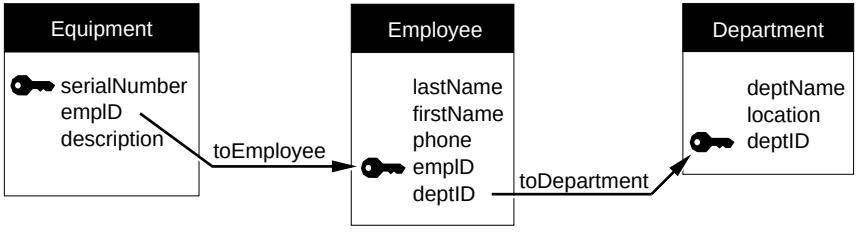


Figure 67. Equipment Allocated by Department

Figure 68 shows how the **Equipment** entity might look after the flattened relationship had been added. In it, **toDepartment** is a relationship defined as **toEmployee.toDepartment**. When your code asks an Equipment object for the value of its toDepartment property, it receives the corresponding Department object. Your code can then query the Department object for the needed properties.

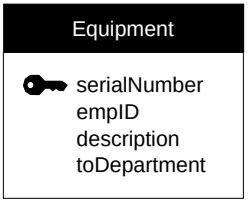


Figure 68. A Flattened Relationship

While the entities involved in a flattened relationship must be related, those relationships can either be to-one or to-many. If any of the relationships are to-many and your code requests the value for a flattened relationship, it will receive an array of objects corresponding to the flattened relationship's destination entity.

Index

E

F, G

P, Q

R

S

T

U

V

W–Z