Chapter 5

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 chapter "Designing Enterprise Objects."

The topics covered in this chapter are as follows:

- How Can I Improve Performance?
- How Do I Generate Primary Keys?
- How Do I Use My Database Server's Integrity-Checking Features?
- How Do I Invoke a Stored Procedure?
- How Do I Order Database Operations?
- How Are Enterprise Objects Deallocated?
- Should I Make Foreign Key Attributes Class Properties?
- How Do I Share Models Across Applications?

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

A simple but effective technique for controlling the number of objects fetched is to implement the EOAdaptorChannel delegate method **adaptorChannel:didFetchRow:** method. In **adaptorChannel:didFetchRow:** you can maintain a count of the objects fetched and take appropriate action when the count reaches a specified limit—for example, you can display an alert panel asking if the user wants to continue fetching, and based on the response you can either fetch the next *N* objects or send the EOAdaptorChannel a **cancelFetch** message.

Faulting

When an EODatabaseContext 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 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.

EODatabaseContext doesn't immediately fetch data for the destination objects of relationships, however, since the related object may never be accessed and fetching is fairly expensive. To avoid this waste of time and resources, destination objects of the class EOFault are created as placeholders. EOFaults come in two varieties: single object faults for to-one relationships, and array faults for to-many relationships.

When an EOFault is accessed (sent a message), it triggers its EODatabaseContext to fetch its data and transform it into an instance of the appropriate object class. 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, which means many 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 an EOFault, 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 doesn't require a trip to the database.

You can set batch faulting in an EOModel. 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 the chapter "Using EOModeler."

To actually control which faults are triggered along with the first one, you can use the EODatabaseContext method **batchFetchRelationship:forSourceObjects:editingContext:**. 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.

Prefetching Relationships

EODatabaseContext defines a hint for use with an EOFetchSpecification in the **objectsWithFetchSpecification:editingContext:** method. Named by the key EOPrefetchingRelationshipHintKey, the hint's value is an NSArray of relationship paths whose destinations should be fetched along with the objects specified. For example, when fetching employees, you can provide a prefetching hint for "department" (and any other relationships) 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.

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 attributes whenever possible.

Flattening attributes has two major drawbacks:

1. The values of flattened attributes can get out of sync with the 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:

```
// 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."

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 set BLOB attributes to be used 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 any such derived data, you can set your EODisplayGroup to refresh its EOAssociations only if its (the EODisplayGroup's) objects were updated.

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:

  ```
  [myDisplayGroup setUsesOptimisticRefresh:YES];
  ```

  This is equivalent to unchecking “Refresh All” in Interface Builder for **myDisplayGroup**.

- Implement the EODisplayGroup delegate method **displayGroup:shouldRedisplayForChangesInEditingContext:** 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, you have to generate and assign unique values to an object’s primary key.

Defining a Primary Key

When designing a database, keep the following tips in mind:

- Don’t use floating point values such as doubles and dates because they aren’t precise in equality tests.
- Use integer primary keys when you want Enterprise Objects Framework to generate primary key values automatically.
- 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" in the "Designing Enterprise Objects" 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. If the primary key value of an object is **nil** 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 **databaseContext:newPrimaryKeyForObject:entity:** that returns a non-**nil** value, the Framework assigns the return value as the object's primary key.

3. A database stored procedure provides a primary key value. If an enterprise object's entity has a stored procedure assigned to the EONextPrimaryKeyProcedureOperation, the Framework invokes the stored procedure and assigns the result as the object's 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:** that provides unique values for primary key attributes.

Note: If the Framework can't assign a primary key using one of the mechanisms above, it raises 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.

If the 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. On the other hand, if an enterprise object generates its own primary key value, 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 **awakeFromInsertionInEditingContext:**.

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 **databaseContext:newPrimaryKeyForObject:entity:**. 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 **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 for the EONextPrimaryKeyProcedureOperation using EOEntity's

setStoredProcedure:forOperation: method, or you can use EOModeler. In the Stored Procedure Inspector, type the name of the stored procedure in the Get PK field. For more information on defining stored procedures and assigning them to entities, see the section "How Do I Invoke a Stored Procedure?".

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:**. 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 m 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 NeXT'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

```
create table eo_sequence_table (
        table_name varchar(32, 0),
        counter integer
)
```

Sybase

```
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

```
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;
```

Sybase

```
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.

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 to 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 **awakeFromInsertionInEditingContext:** method. For example:

```
- (void)awakeFromInsertionInEditingContext:(EOEditingContext *)context
{
    [super awakeFromInsertionInEditingContext:context];
    // 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."

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 now empty department, then saves all changes. Enterprise Objects Framework analyzes the object graph to determine what operations have taken place. It generates and executes database operations to update the database accordingly. Enterprise Objects Framework does not guarantee that the operations will be ordered to observe relational integrity rules. In other words, it is possible for the deletion of the Department object to occur before all the employees are updated to reflect their new department. In this case, the database will not allow the deletion of the department, and the whole set of operations fail as a result.

To avoid sequencing problems of this type, implement an EODatabaseChannel delegate method to order database operations before they are sent to the database. For more information, see the section “How Do I Order Database Operations?”.

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. Depending on what a stored procedure does, you can either invoke it explicitly or specify that the Framework invoke it for common database operations.

Defining a Stored Procedure

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 in your model. 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. You can also add a stored procedure definition to an existing model.

To add a stored procedure in EOModeler:

1. Choose Tools m Stored Procedures.

2. Select the model icon.

3. Choose Property m Add Stored Procedure.

4. Specify a name and external name for the stored procedure.

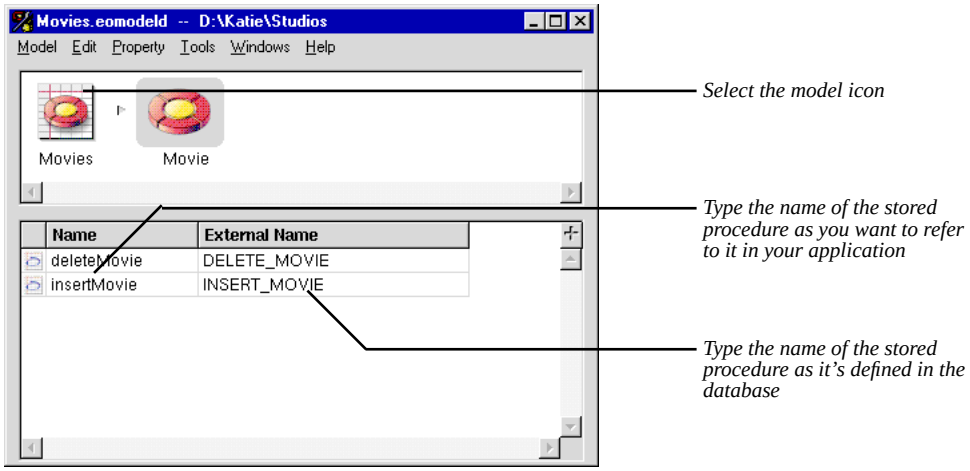

Figure 48. *Adding a Stored Procedure*

You must also define an *argument* for a stored procedure's return value and for each of its parameters. Add arguments to a stored procedure the same way you add attributes to an entity. In fact, the arguments of a stored procedure are represented with EOAttribute objects.

Note: The Advanced Attribute Inspector isn't applicable to stored procedure arguments. As a result, you can't access it while editing an argument.

To define and display the attributes of a stored procedure:

1. Double-click the icon to the left of a stored procedure in the stored procedures view of the Model Editor.

2. Choose Property m Add Argument.

3. Use the menu to add columns for the stored procedure. By default, the table mode of the Model Editor has just four columns for stored procedure arguments: Name, Value Class, External Type, and Width. The menu provides these additional columns: Column, Direction, Precision, Scale, Value Type.

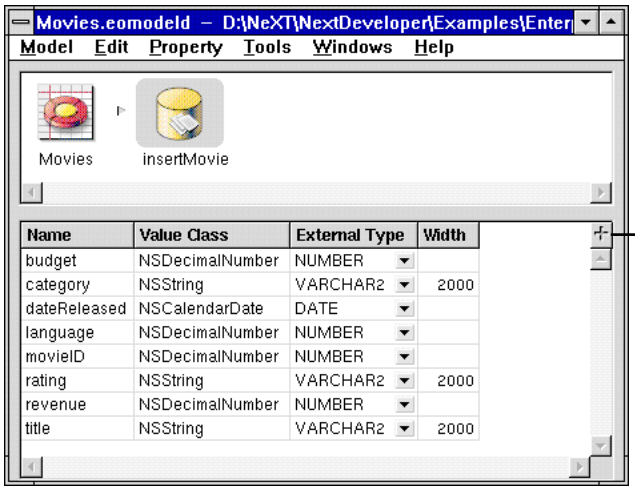

Figure 49. *Adding a Column to the Stored Procedure Argument View*

4. Specify the argument's characteristics. Minimally, you must provide a name, a column, a direction, an external type, and value class for each argument. The following table describes the characteristics you can set for a stored procedure argument.

Characteristic	What it is
Column	The name of a parameter as it is defined in the database (doesn't apply to a "returnValue" argument).
Direction	In, InOut, Out, or Void. Don't choose Void; it's reserved for future use.
External Type	The data type of the argument as it's defined in the database.
Name	The name your application uses for the argument.
Precision	The number of significant digits (applies to number data only).
Scale	The number of digits to the right of the decimal point (applies to number data only).
Value Class	The Objective-C type to which the argument value will be coerced in your application.
Value Type	The format type for custom value classes such as "TIFF" or "RTF".
Width	The maximum width (applies to string, raw, and binary data).

For example, to add arguments for the Sybase stored procedure defined as:

```
create proc movie_by_date (@begin datetime, @end datetime) as
begin
    select
        CATEGORY, DATE_RELEASED, LANGUAGE, MOVIE_ID, RATING,
        REVENUE, STUDIO_ID, TITLE
    from MOVIES
    where DATE_RELEASED > @begin and DATE_RELEASED < @end
end
```

you would add an argument for **@begin** and **@end** with column names “begin” and “end”, respectively.

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 the Framework invokes your stored procedure automatically, the argument names of a stored procedure must match the name of a corresponding EOAttribute object. For example, if you want to invoke a stored procedure whenever the Framework fetches a Movie object by its primary key, the stored procedure’s argument names must correspond to the primary key attributes of the Movie entity. The section “Invoking a Stored Procedure Automatically” discusses this requirement more thoroughly.

Invoking a Stored Procedure Automatically

You can define stored procedures to perform the following operations:

- **EOFetchAllProcedureOperation** fetches all the objects for an entity.
- **EOFetchWithPrimaryKeyProcedureOperation** fetches an object by its primary key.
- **EOInsertProcedureOperation** inserts a new object.
- **EODeleteProcedureOperation** deletes an object.
- **EONextPrimaryKeyProcedureOperation** generates a new primary key value.

If you associate 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 as described above.

3. Associate the stored procedure with the Customer entity’s insert operation.

You can associate a stored procedure with an entity using EOModeler or you can do it programmatically using EOEntity's **setStoredProcedure:forOperation:** method.

In EOModeler:

1. Select the entity with which you want to associate a stored procedure.
2. Choose Tools m Inspector.
3. Click the Stored Procedures Inspector icon.

4. Type the name of the stored procedure in the field associated with the appropriate database operation.

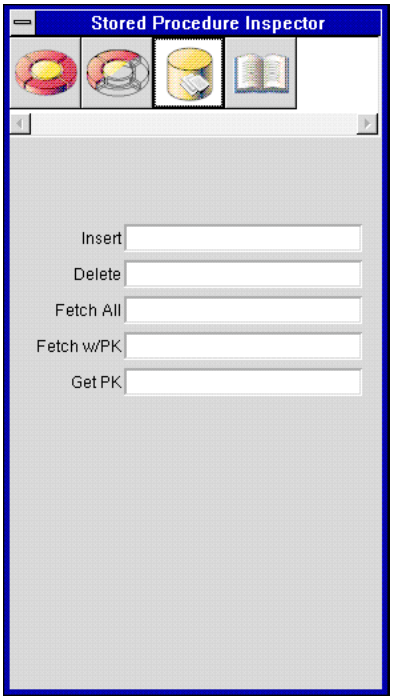

Figure 50. *The Stored Procedure Inspector*

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:

EOFetchAllProcedureOperation

The EOFetchAllProcedureOperation 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 EOFetchAllProcedureOperation stored procedure has a return value, Enterprise Objects Framework ignores it.

EOFetchWithPrimaryKeyProcedureOperation

The EOFetchWithPrimaryKeyProcedureOperation 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 EOFetchWithPrimaryKeyProcedureOperation 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 EOFetchAllProcedureOperation stored procedure.

If an EOFetchWithPrimaryKeyProcedureOperation stored procedure has a return value, Enterprise Objects Framework ignores it.

EOInsertProcedureOperation

The EOInsertProcedureOperation 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 EOInsertProcedureOperation stored procedure should not return a result set. Also, if an EOInsertProcedureOperation stored procedure has a return value, Enterprise Objects Framework ignores it.

EODeleteProcedureOperation

The EODeleteProcedureOperation 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 EOFetchWithPrimaryKeyProcedureOperation stored procedures.

An EODeleteProcedureOperation stored procedure should not return a result set. Also, if an EODeleteProcedureOperation stored procedure has a return value, Enterprise Objects Framework ignores it.

EONextPrimaryKeyProcedureOperation

The EONextPrimaryKeyProcedureOperation 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 EOFetchWithPrimaryKeyProcedureOperation stored procedures.

An EONextPrimaryKeyProcedureOperation stored procedure should not return a result set. Also, if an EONextPrimaryKeyProcedureOperation 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:

```
EOAdaptorChannel *adChannel;      // Assume this exists.
EOStoredProcedure *sumOfRevenue;
NSDictionary *results;

sumOfRevenue = [model 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.

If you want to invoke a stored procedure that returns rows, you use **fetchRowWithZone:** as you would if you were fetching the results of a **selectAttributes:fetchSpecification:lock:entity:** message. For example, the following code excerpt fetches Movie objects using the **fetchAllMovies** stored procedure:

```
EOAdaptorChannel *adChannel;      // Assume this exists.
EOStoredProcedure *fetchAllMovies;
NSDictionary *row;

fetchAllMovies = [model storedProcedureNamed:@"fetchAllMovies"];
[adChannel executeStoredProcedure:fetchAllMovies 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:withValues:** message. For example, the following code excerpt uses a stored procedure to insert a row into the database:

```
EOAdaptorChannel *adChannel;  // Assume this exists.
EOStoredProcedure *insert;
NSDictionary *row;            // Assume this contains the values
                              // for the row that's being inserted.

insert = [model storedProcedureNamed:@"insertTest"];
[adChannel executeStoredProcedure:insert withValues:row];
```

For more information on invoking stored procedures explicitly, see the EOAdaptorChannel class specification.

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.

By default, an EODatabaseContext sorts database operations alphabetically based on entity name. For each entity, lock operations are ordered first, followed by insert, update, delete, and stored procedure operations. If you use referential integrity-checking features of your database, you might need to order the operations differently.

For example, if your database requires that each foreign key in a newly inserted row correspond to an existing row, you can't insert a Guest row unless it references a valid Member row. If a user creates a new member and a new guest for that member, you have to order the database operations so the member is inserted before the guest.

Another reason you might need to order database operations is to improve concurrency. If a database implements page- or table-level locking, you are more likely to encounter deadlocks when multiple, simultaneous transactions don't lock tables in the same order.

You can order database operations by implementing either or both of the EODatabaseContext delegate methods

– databaseContext:willOrderAdaptorOperationsFromDatabaseOperations:
– databaseContext:willPerformAdaptorOperations:adaptorChannel:

The former method provides the delegate with more information than the latter. The second argument to the "willOrder" method is an array of EODatabaseOperation objects that reference the enterprise objects on which to operate and also include the globalIDs of these objects. The "willPerform" method, however, is more convenient to use. 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.

How Are Enterprise Objects Deallocated?

If you use an EODisplayGroup to fetch enterprise objects into your application, you might wonder:

- Who “owns” the objects?
- How do they get deallocated?
- How are their snapshots deallocated?
- What happens if you have retain cycles?

In applications that fetch a lot of enterprise objects or are long-running, these are important questions. However, you don’t have to worry about answering most of them. As long as you follow the object ownership conventions defined in the Foundation framework, enterprise objects and their related resources are automatically deallocated when they are no longer in use. For more information on this automatic object disposal mechanism, see the introduction to the *Foundation Framework Reference.*

Who Owns an Enterprise Object?

In design terms, one object might own another; but in OpenStep programming terms, 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
{
    [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.

invalidateAllObjects 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?

No. You shouldn't 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:

```
- (id)departmentID
{
    return ValueForPrimaryKey(department, @"departmentID");
}

id ValueForPrimaryKey(id object, NSString *key)
{
    if ([EOFault isFault:object]) {
        EOKeyGlobalID *globalID;
        EOEntity *entity;

        globalID = [[object editingContext] globalIDForObject:object];
        entity = [[EOModelGroup defaultGroup]
                entityNamed:[globalID entityName]];
        return [[entity primaryKeyForGlobalID:globalID] valueForKey:key];
    } else {
        return [object valueForKey:key];
    }
}
```

The function **ValueForPrimaryKey** verifies that the destination object is not an EOFault object before accessing its primary key value. If you don't do this verification, you may unnecessarily trigger a fetch of the destination object.

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 and InterfaceBuilder). When you use this approach, you should also put the enterprise object classes that correspond to the model in the framework.

For Enterprise Objects Framework to find a model in a framework, that framework must be built and installed. Further, even at design time 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 InterfaceBuilder 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 defaults command (executed in a shell):

```
defaults write NSGlobalDomain
EOProjectSourceSearchPath"($(HOME)/myProjectsDirectory1,
/myOtherProjectsDirectory)"
```

Then, when Interface Builder or EOModeler looks 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.