Chapter 2

Designing Enterprise Objects

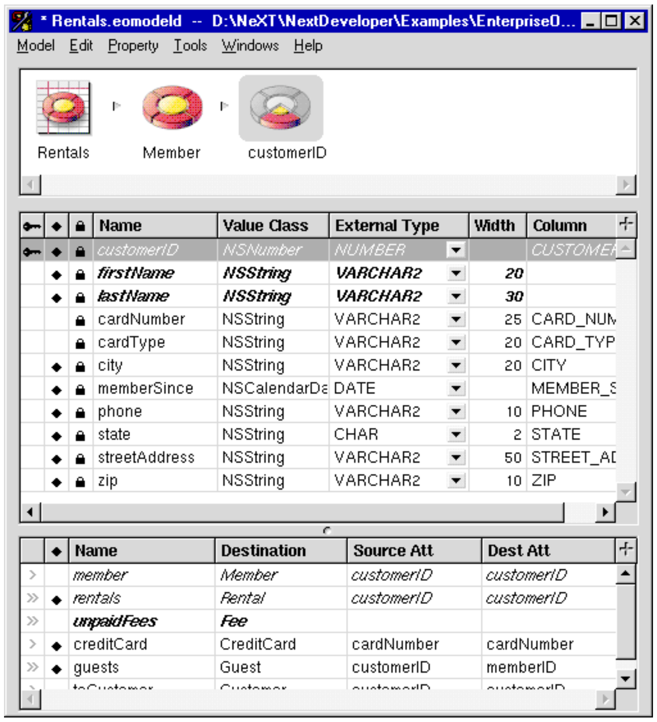
The Enterprise Objects Framework and the applications you build with it revolve around the enterprise objects that you design. Designing these objects, then, is in many ways the essence of creating an Enterprise Objects Framework application. This chapter explains the mechanics of designing enterprise 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 stage. It 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 Member entity in the Rentals database. A member is a video store customer who's authorized to rent videos and whose guests may also rent videos.

Figure 1 shows the Member entity, as constructed in EOModeler. The top part of the window displays Member's attributes, and the lower part of the window displays its relationships. For more discussion of EOModeler, see the chapter "Using EOModeler."


* Rentals.eomodeld -- D:\NeXT\NextDeveloper\Examples\EnterpriseO...
Model Edit Property Tools Windows Help
Rentals
Member
customerID

🔑	◆	🔒	Name	Value Class	External Type	Width	Column
🔑	◆	🔒	*customerID*	*NSNumber*	*NUMBER*		*CUSTOME*
	◆	🔒	***firstName***	***NSString***	***VARCHAR2***	***20***	
	◆	🔒	***lastName***	***NSString***	***VARCHAR2***	***30***	
		🔒	cardNumber	NSString	VARCHAR2	25	CARD_NUN
		🔒	cardType	NSString	VARCHAR2	20	CARD_TYP
	◆	🔒	city	NSString	VARCHAR2	20	CITY
	◆	🔒	memberSince	NSCalendarDa	DATE		MEMBER_S
	◆	🔒	phone	NSString	VARCHAR2	10	PHONE
	◆	🔒	state	NSString	CHAR	2	STATE
	◆	🔒	streetAddress	NSString	VARCHAR2	50	STREET_AD
	◆	🔒	zip	NSString	VARCHAR2	10	ZIP

	◆	Name	Destination	Source Att	Dest Att
>		*member*	*Member*	*customerID*	*customerID*
>>	◆	*rentals*	*Rental*	*customerID*	*customerID*
>>		***unpaidFees***	***Fee***		
>	◆	creditCard	CreditCard	cardNumber	cardNumber
>>	◆	guests	Guest	customerID	memberID

Figure 1. *The Member Entity in EOModeler*

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 75 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 Member class in the Rentals database has specialized behavior—for example, it validates changes to objects, calculates the cost restrictions of a member's guests, and keeps track of a member's rentals—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 template code 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. The Member class flattens two properties from Customer: **firstName** and **lastName**. These flattened attributes are included in Member as class properties, but the relationship to Customer that Member uses to flatten the attributes doesn't need to be a class property since it's otherwise not needed. For more discussion of this topic see "How Should Your Enterprise Object Manage Relationships with Other Objects?" on page 68.
- Relationships that represent an entity's relationship to an intermediate join 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 Role 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

Role, 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 classes:

- NSString
- NSCalendarDate
- NSDecimalNumber
- NSNumber
- NSData
- EONull (representing NULL values on the server only)
- 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.

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 NSDecimalNumbers. This is because in the Enterprise Objects Framework, numeric database values are stored in one of two types of objects: NSNumber or NSDecimalNumber. When you use NSNumber, values are limited to double precision and operations are inexact. Using NSDecimalNumber provides high precision (38 decimal digits) and smooth conversions between strings, NSDecimalNumbers, and money.

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

Conversion of Numeric Values

When the Framework passes an NSNumber value to your object, the value is converted to the C scalar (numeric) type required by your accessor method or instance variable (for more information, see "Accessing an Enterprise Object's Data" on page 75). For example, suppose your enterprise object defines these accessor methods:

– (void)setAge:(int)*age*
– (int)age

For the **setAge:** method, the NSNumber 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 NSNumber.

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 pointers in the object graph and to-many relationships are represented as NSArrays. You can see this more clearly if you look at the way the to-one relationship to CreditCard and the to-many relationship to Guest are represented as instance variables in the Member class:

```
// Pointer to a CreditCard object in the object graph
id creditCard;

// Array of Guest objects
NSMutableArray *guests;
```

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 the member's **creditCard** relationship to access the **authorizationDate** property in the CreditCard object:

```
date = [[member creditCard] authorizationDate];
```

Likewise, the following statement uses the member's **guests** relationship to return an NSArray containing the member's Guest objects:

```
guestArray = [member guests];
```

Referential Integrity

When your enterprise object has relationships with other objects, you can use EOModeler to define rules to govern those relationships:

- 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 pointer 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)

  In addition, 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 pointer is set to **nil**. If you really intend to delete the employee from the database, you'd have to do it explicitly.

  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.

For more information on how to set these options, see the chapter "Using EOModeler."

Mapping an Entity Across Multiple Tables

In certain special cases you may decide to use EOModeler to "flatten" the attributes of a destination entity. When you flatten an attribute, you effectively add it from the destination entity to the source entity. 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:

- Implementing vertical inheritance mapping.

  For example, the Member class 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, see "Inheritance" on page 83.

- 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.

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

Note: 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 chapter "Using EOModeler."

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. For more discussion of the different approaches you can use for inheritance, see "Inheritance" on page 83.

Implementing an Enterprise Object

As discussed in the section "EOGenericRecord or Custom Class?" on page 65, 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. EOGenericRecord is the default enterprise object class provided in the Enterprise Objects Framework. Unlike a custom class, it implements no custom behavior.

For both EOGenericRecords and custom classes, the Enterprise Objects Framework provides mechanisms for the following:

- Accessing data

  For accessing data in enterprise objects, the Enterprise Objects Framework provides the informal protocols EOKeyValueCoding and EOKeyRelationshipManipulation. For more discussion of these protocols, see "Writing Accessor Methods" on page 73.

- 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 EOAdaptorChannel's **primaryKeyForNewRowWithEntity:**. If you need to provide a custom mechanism for assigning primary keys, you can implement the EODatabaseContext delegate method **databaseContext:newPrimaryKeyForObject:entity:**. Using these two techniques, you don't need to store the primary key in your enterprise object.

From this point on, the assumption is that you're using custom enterprise object classes. The following sections describe how to add behavior to your custom classes.

Generating Template Source Code

When you use the Create Template command in EOModeler to generate template source code for a custom class, the header (**.h**) and implementation (**.m**) files that are created contain the default implementation for your enterprise object class.

Looking at header file **Member.h** shows you the instance variables and accessor methods that are automatically created for you when you generate template source code for the Member class.

Member.h

```
#import <EOControl/EOControl.h>

@class Guest;
@class CreditCard;
@class Rental;

@interface Member : NSObject
{
    int customerID;
    NSString *city;
    NSCalendarDate *memberSince;
    NSString *phone;
    NSString *state;
    NSString *streetAddress;
    NSString *zip;
    NSString *firstName;
    NSString *lastName;
    CreditCard *creditCard;
    NSMutableArray *rentals;
    NSMutableArray *guests;
}

- (void)setCustomerID:(int) value;
- (int) customerID;

- (void)setCity:(NSString *)value;
- (NSString *)city;

- (void)setMemberSince:(NSCalendarDate *)value;
- (NSCalendarDate *)memberSince;

- (void)setPhone:(NSString *)value;
- (NSString *)phone;

- (void)setState:(NSString *)value;
- (NSString *)state;
- (void)setStreetAddress:(NSString *)value;
- (NSString *)streetAddress;

- (void)setZip:(NSString *)value;
- (NSString *)zip;

- (void)setFirstName:(NSString *)value;
- (NSString *)firstName;

- (void)setLastName:(NSString *)value;
- (NSString *)lastName;

- (void)setCreditCard:(CreditCard *)value;
- (CreditCard *)creditCard;

- (NSArray *)rentals;
- (void)addToRentals:(Rental *)object;
- (void)removeFromRentals:(Rental *)object;

- (NSArray *)guests;
- (void)addToGuests:(Guest *)object;
- (void)removeFromGuests:(Guest *)object;


@end
```

The template source code for Member illustrates some of the basic principles involved in implementing an enterprise object. These principles are discussed in the following sections.

Writing Accessor Methods

When you generate template source code for a custom class in EOModeler, the resulting **.m** file includes default accessor methods. Accessor methods let you set and return the values of your class properties (instance variables). For example, here are some of the accessor methods in the **Member.m** file:

```
- (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; }

    - (NSArray *)rentals { return rentals; }

    - (void)addToGuests:(Guest *)object
    {
        // a to-many relationship
        [self willChange];
        [guests addObject:object];
    }
    - (void)removeFromGuests:(Guest *)object
    {
        // a to-many relationship
        [self willChange];
        [guests removeObject:object];
    }
    - (NSArray *)guests { return guests; }
```

Features introduced by this code excerpt are discussed in the following sections.

Change Notification

In the above code excerpt from **Member.m**, you can see that each of the "set" methods includes the statement `[self willChange]`.

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 template source code's "set" methods, but whenever you add your own methods that change the object's state, you need to remember to include `[self willChange]`.

Accessing Data through Relationships

In the Member class, note that relationships and flattened properties are treated no differently than properties based on the entity's original attributes.

For example, Member 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 Member's credit card, your code can do something like this:

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

Note that you can modify attributes in the object graph regardless of what table they came from. Member's **setFirstName:** method lets you modify the **firstName** flattened attribute, even though it's actually 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, with a default implementation that takes advantage of methods you're likely to write for your own use. This interface is defined by the EOKeyValueCoding informal protocol, which abstracts object access in terms of key-value pairs.

In accessing an object's class properties, the default implementations of the key-value coding methods use the class definition as follows:

1. The key-value coding method looks for an accessor method based on the property name. For example, with a property named **lastName**, key-value coding looks for a "set" method of the form **setLastName:** (note that the first letter of the property name is made uppercase), and a "get" method of the form **lastName**.

2. If the key-value coding method doesn't find an accessor method, it looks for an instance variable whose name is the same as the property's and sets or retrieves its value directly.

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 preserve the encapsulation of data enabled by object-oriented programming, allowing 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 77.

You can also use an object's relationship properties to access the data in other objects. 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];
```

You can traverse multiple objects. For example, suppose the Department object has a relationship to a Facility object. From Employee, you can access the properties in Facility as follows:

```
// Get the location of the Employee’s department
[[[employee department] facility] location];

// Change the location of the employee’s department
[[[employee department] facility] setLocation:newLocation];
```

Writing Derived Methods

The template source code generated by EOModeler provides a basic implementation that doesn’t go beyond the functionality provided by an EOGenericRecord. But you can use the template 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 perform computations based on the values of class properties. For example, members have an overall credit limit, and their guests have a cost restriction. To calculate the total cost restrictions of all of a member’s guests, you can have a method in **Member.m** like the following:

```
- (NSDecimalNumber *)totalRestrictions {
    NSDecimal result = [[NSDecimalNumber zero] decimalValue];
    int i = [guests count];
    while( i-- ) {
        NSDecimalNumber *restriction = [[guests objectAtIndex:i]
             valueForKey:@"costRestriction"];
        if(restriction) {
           NSDecimal total;
           total = [restriction decimalValue];
       (void)NSDecimalAdd(&result, &result, &total, NSRoundBankers);
        }
    }
    return [NSDecimalNumber decimalNumberWithDecimal:result];
}
```

The instance variable **guests** is a property of the Member class. It represents a to-many relationship to the Guest object. As the preceding code excerpt implies, you can access data in Guests transparently. The Enterprise Objects Framework simply traverses the object graph and performs a calculation on the **costRestriction** class property of the Member’s Guest objects.

Performing Validation

As described in the previous section, members have a credit limit and their guests have a cost restriction. The total cost restrictions of a member’s guests can’t exceed the member’s total credit limit. When users of your application add a new guest for a member, you probably want to make sure that adding the guest’s cost restriction to those of other guests doesn’t push the total up over the member’s credit limit. The following code excerpt shows one way this might be achieved:

```
// Calculate how much is left in the member’s credit limit after
// subtracting the total cost restrictions of the member’s guests
- (NSDecimalNumber *)restrictionPool {
    return [[creditCard limit] decimalNumberBySubtracting:
       [self totalRestrictions]];
}

// Perform validation before a guest is inserted or updated in
// the database.
// If adding or modifying the guest breaks the member’s credit limit,
// return an exception.
- (NSException *)validateForSave
{
    if ([[self restrictionPool] doubleValue] < 0)
       return [NSException validationExceptionWithFormat:@"The cost
              restrictions on the guests exceed the credit limit"];
    // pass the check on to the EOClassDescription
    return [super validateForSave];
}
```

The **validateForSave** method is part of a category of NSObject that uses the EOClassDescription class to provide default implementations of validation methods. These methods are invoked automatically by Framework components such as EODisplayGroup and EOEditingContext. They are:

- validateForSave
- validateForDelete
- validateForInsert
- validateForUpdate
- validateValue:forKey:

The EOClassDescription class effectively extends the behavior of enterprise objects. EOClassDescription usually derives validation rules (such as NULL constraints and referential integrity rules) from an EOEntity in a model file. When you perform a particular operation on an enterprise object (such as attempting to delete it), the EOEditingContext sends these validation messages to your enterprise object, which in turn (by default) forwards them to

EOClassDescription. 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 an department object that has employees. If a user attempts to delete a department that has employees, an exception is returned and the deletion is refused.

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 vacation data for an employee, you might want to confirm that the start date precedes the end date before committing the changes.

However, you can also implement validation methods for individual class properties. Such methods take the form **validate***Property*. For example, the Member class might have a **validateStreetAddress** method. If a user tries to assign a value to **streetAddress**, the default implementation of **validateValue:ForKey:** invokes the **validateStreetAddress:** method. Based on the result, the operation is either accepted or refused.

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.

You can design your application 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***Key*: 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.

Creating and Inserting Objects

In an Enterprise Objects Framework application, when new enterprise objects are inserted into the database, it's often through an EODisplayGroup (assuming the application uses the interface layer). However, it's also common to create and insert an enterprise object programmatically—either because your application isn't using the interface layer, or because you're creating and inserting objects as the by-product of another operation.

You don't have to do anything special to create an instance of an enterprise object. You can create it using the **alloc** and **init** methods just like any other object. Once you create the object, you insert it into an EOEditingContext using EOEditingContext's **insertObject:** method.

For example, the following code excerpt from **Customer.m** in the on-line examples creates Fee and Rental enterprise objects as the by-product of a customer (a Guest or a Member) renting a unit at a video store. Once the objects have been created, they're inserted into the current enterprise object's EOEditingContext.

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

    // Create Rental and Fee objects
    rental = [[[Rental alloc] init] autorelease];
    fee = [[[Fee alloc] init] autorelease];

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

    // Insert the two objects into the current object's editingContext.
    [[self editingContext] insertObject:rental];
    [[self editingContext] insertObject:fee];
}
```

Working with Relationships

In the code excerpt shown in the preceding section, notice that before the objects are inserted into the EOEditingContext, the method **addObject:toBothSidesOfRelationshipWithKey:** is used. This method is part of the EOKeyRelationshipManipulation informal protocol. EOKeyRelationshipManipulation provides methods for manipulating an enterprise object's relationship properties.

addObject:toBothSidesOfRelationshipWithKey: is used to manage reciprocal relationships, in which the destination entity of a relationship has a back pointer to the source. For example, Fee and Unit both have back pointers to Rental, and Rental has a back pointer 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 pointer 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 pointers 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 **addObject:toBothSidesOfRelationshipWithKey:**. 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.

In addition to the **addObject:toBothSidesOfRelationshipWithKey:** method, EOKeyRelationshipManipulation defines the following methods:

- **addObject:toPropertyWithKey:** adds a specified object to the relationship property (NSArray) with the specified name (key). This method is the primitive used by **addObject:toBothSidesOfRelationshipWithKey:**.

In accessing an object's class property, the default implementation of EOKeyRelationshipManipulation 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, **addObject:toPropertyWithKey:** looks for a property called **projects**. If it finds it, it adds it 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 **nil**, a new array is created and set on the object.

- **removeObject:fromPropertyWithKey:** removes a specified object from the relationship property (NSArray) with the specified name (key).

  This method follows the same pattern as **addObject:toPropertyWithKey:**. That is, it looks for a selector of the form **removeFromProjects:**, then for a property called **projects**. If neither of these conditions is met, it raises an exception. If it finds the property but it doesn't contain the specified object, this method simply returns.

- **removeObject:fromBothSidesOfRelationshipWithKey:** removes a specified object from both sides of a relationship property with the specified name (key). Like **addObject:toBothSidesOfRelationshipWithKey:**, 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:**. 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 **awakeFromInsertionInEditingContext:** in the Member class sets the current date as the value of the **memberSince** property:

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

You use the **awakeFromInsertionInEditingContext:** 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 **awakeFromFetchInEditingContext:**, 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 database channel is still busy fetching when an enterprise object receives **awakeFromFetchInEditingContext:**, the object must be careful about sending messages to other enterprise objects, since they may be faults. For more information about faults, see the EOFault class specification in the *Enterprise Objects Framework Reference*.

Initializing Enterprise Objects

An enterprise object is typically initialized with **init**; all of its class property values are handled through the key-value coding methods, so no special initialization is usually needed. You don't generally use **init** to assign default values to your enterprise objects since **init** is invoked to initialize instances of your class being fetched from the database. Any values you assign in **init** 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. If an enterprise object responds to the **initWithEditingContext:classDescription:globalID:** method, EODatabaseContext 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 want the ability to give all of a member's guests a boost in their rental limits. You could implement a method such as the following in **Member.m**:

```
- (void)boostRestrictions
{
    int i;
    NSDecimalNumber *rate = [NSDecimalNumber
        decimalNumberWithString:@"1.50"];

    for (i = [guests count] - 1; i >= 0; i--) {
        Guest *guest = [guests objectAtIndex:i];
        [guest setCostRestriction:[[guest costRestriction]
                decimalNumberByMultiplyingBy:rate]];
    }
}
```

Of course, when a user invokes this method and attempts to save the new values to the database, the **validateForSave** method described in "Performing Validation" on page 77 will be invoked. If the total amount of the guests' cost restrictions exceeds the member's credit limit, the operation is refused.

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 managed by the EOEditingContext, and the EOEditingContext 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.

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.

Enterprise Objects Framework maps an object-oriented model to a relational database model. While Enterprise Objects Framework largely protects the programmer, and especially the user, from having to know a great deal about relational databases, this knowledge becomes necessary when determining the storage and retrieval of a class hierarchy.

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.

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.

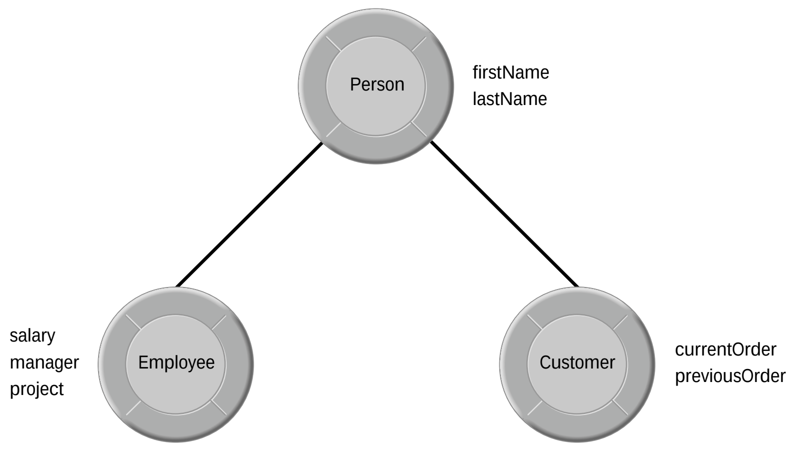


Figure 2. *Object Hierarchy*

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 hierarchy?

  When an object 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 hierarchy (Employees and Customers).

- How deep is the hierarchy?
- 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 even 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 against your object hierarchy. 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 object 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 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 has the attributes that are introduced by that class.

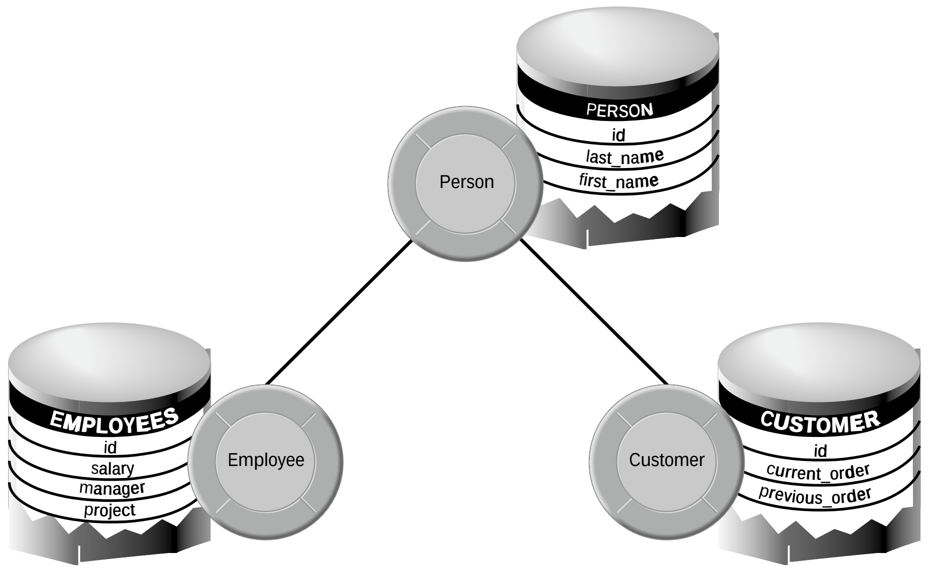


Figure 3. *Vertical Inheritance Mapping*

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

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

1. In EOModeler, create a child entity mapped to its own table.
2. Add a relationship between the child and the parent table.
3. If the parent entity is abstract, set it as such.
4. Flatten the parent attributes into the child.
5. Set the parent entity for each of the child entities.

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. 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.

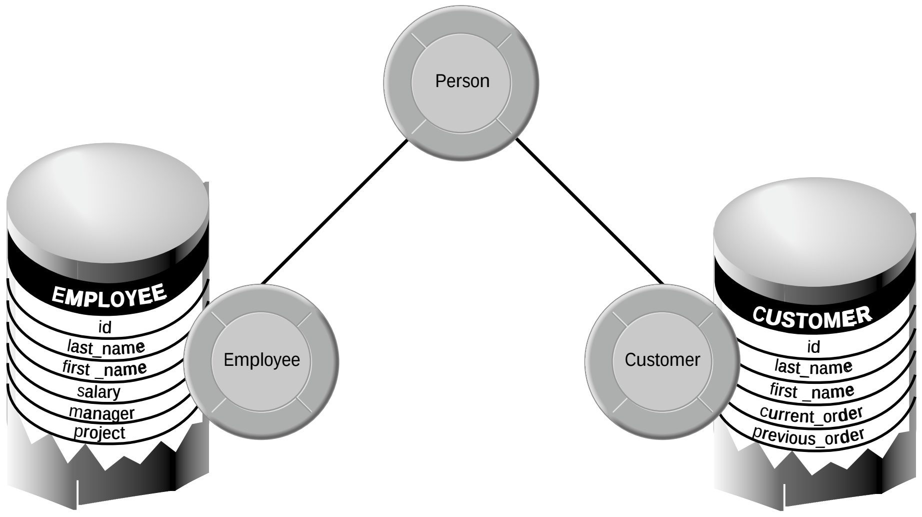


Figure 4. *Horizontal Inheritance Mapping*

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. In EOModeler, set Person as the parent entity of Employee and Customer.
2. If Person is an abstract entity, set it as such in the Inspector. Under horizontal mapping, if Person doesn't have its own table (that is, if you never fetch Person objects that aren't Employees or Customers), then Person is 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.

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).

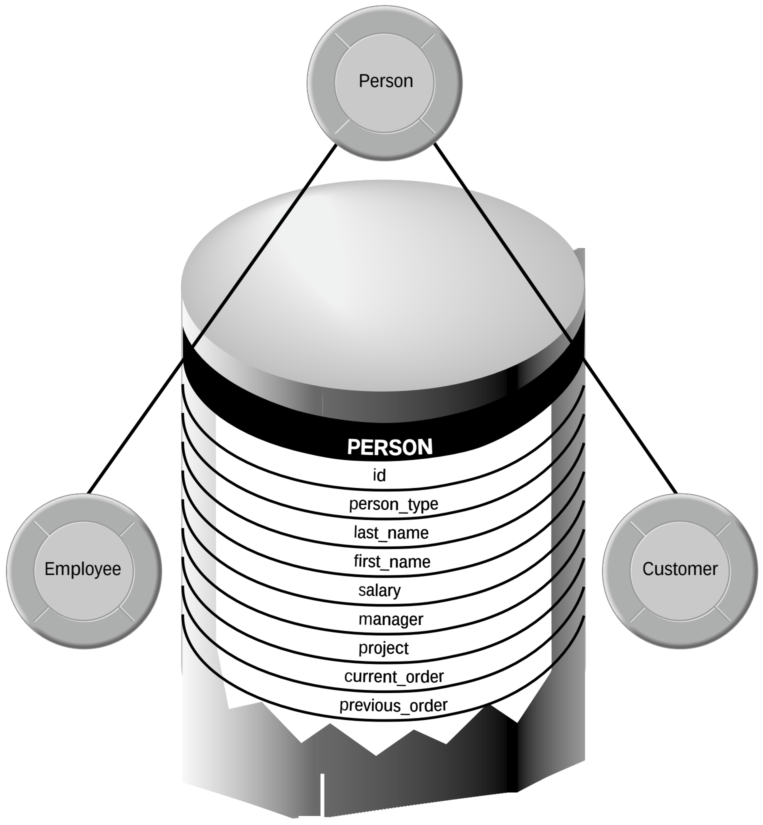


Figure 5. *Single Table Mapping*

Single table mapping can consume an inordinate amount of space since every row includes columns for every one of the other entities' attributes. Whether you decide to use this approach may depend on how your database stores NULLs and other data types. 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.

Creating an EOModel for Single Table Mapping

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

1. In EOModeler, set Person as the parent entity of Employee and Customer.

2. If Person is an abstract entity, set it as such in the Inspector.

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.

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 hierarchy, n equals 3.

Advantages and Disadvantages of Mapping Approaches

This section discusses the advantages and disadvantages of each of the inheritance mapping approaches.

Vertical Mapping

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 hierarchy. The primary virtue of this approach, however, is its clean, "normalized" design.

Disadvantages. Vertical mapping is the least efficient of all of the approaches. Every layer of the 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

Advantages. 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 hierarchy.

This approach works well for deep class hierarchies, as long as the fetch occurs against the leaves of the hierarchy (Employee and Customer) rather than against the root (Person). When a root fetch is used in this scenario, it's more efficient than vertical mapping (since no joins are performed) and less efficient than one table mapping (since it requires more fetches). However, if you typically only fetch one type of leaf class at a time, this approach can actually be more efficient than single table mapping. This is because with single table mapping, you have to look at all of the rows in the database to find the ones needed to instantiate objects of a particular type.

Disadvantages. This technique is less effective if both Customers and Employees have to be fetched at the same time, since both tables then need to be accessed.

Another problem occurs when attributes need to be added to the Person superclass. The number of tables that need to be altered equals the number of subclasses—the more subclasses you have, the more effort is required to maintain the superclass. However, table maintenance happens far less often than fetches, so it should be weighted in importance accordingly.

Single Table Mapping

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. This approach can use an inordinate amount of storage space. 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 typically just fetches one type of leaf object at a time.)

General Guidelines

While deep 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 object hierarchy onto a relational database, the result is likely to be poor performance and a database that's difficult to maintain.

Enterprise Objects Framework doesn't support mapping inheritance hierarchies across tables in separate databases. Instead, you can set up groups of objects that cooperate across databases, in which related objects forward messages to each other.

Fetching and Inheritance

Once you've designed your class hierarchy and set up your EOModel to support that 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.

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, fetching Persons shallowly fetches only Persons matching the specified qualifier, while fetching Persons deeply also fetches all Employees and Customers matching the specified qualifier.

You can control this behavior by using EOFetchSpecification's **setIsDeep:** method. This method specifies whether a fetch should include sub-entities of the fetch specification's entity. If this method is set to YES, sub-entities are also fetched; if it's set to 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:**, 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 can use a restricting qualifier to fetch 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:

entity:relationshipForRow:relationship:

This method 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:primaryKey:isFinal:

This method 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. 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 to YES.

entity:classForObjectWithGlobalID:

This method 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.

Gotchas

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

Numeric Values and NULL

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

unableToSetNilForKey: is part of the EOKeyValueCoding protocol, and you use it to set policy for an attempt to assign **nil** to an instance variable that requires a C scalar type. Classes can implement this method to determine the behavior when **nil** is assigned to a property in an enterprise object that requires a C scalar type (such as **int** or **float**). One possible implementation is to reinvoke **takeValue:forKey:** with a special constant value (such as `[NSNumber numberWithInt:0]`).

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 EONull objects, but they're converted to **nil** 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 **id** value received through an accessor method to see whether it's **nil** before sending it a message.

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 an EODatabaseContext creates an enterprise object, it creates fault objects for related objects, and these fault objects can be passed to your enterprise objects in a key-value coding message while the database channel 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 with the busy database channel, resulting in resource contention.

Don't Override isEqual:

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