Enterprise Objects Framework Tools And Techniques

This manual describes WebObjects 4.0.

Writing: Kelly Toshach
Graphic Design: Karin Stroud
Production: Terri Fitzmaurice
Release Control: Chona Reyes

Table of Contents

Introduction

Introduction to Enterprise Objects Framework Tools and Techniques

One of the ingredients in all Enterprise Objects Framework applications is a *model*. A model defines, in entity-relationship terms, the mapping between your enterprise objects and a relational database. The EOModeler application, being the tool you use to create and maintain models, has a prominent place in the suite of applications you use to develop your Enterprise Objects Framework applications.

You use EOModeler to:

- Read the data dictionary from a database to create a default model, which can then be tailored to suit the needs of your application.
- Specify enterprise object classes for the entities in your model.
- Generate source code files for the enterprise object classes you specify.
- Define fetch specifications (queries) that you can invoke by name in your applications.
- Create and drop databases and database tables.

This book describes EOModeler and the techniques you use to define models that work effectively in Enterprise Objects Framework applications.

For more general information on Enterprise Objects Framework and on creating applications that use it, see the book *Enterprise Objects Framework Developer's Guide*.

Chapter 1

Creating a New Model

This chapter describes how to create a new model from an existing database. It is organized into the following sections:

About Models

Although a model can be generated dynamically at run time, you typically create models using EOModeler and then add them to your project as model files.

Models are designed to be loaded incrementally to maximize performance. A model consists of one global file, with a separate file for each entity. Entity descriptions are loaded in to an application as needed. Models have an **.eomodeld** file wrapper (which is actually a directory), and the individual entity files within the model are in ASCII format. If you want to view the ASCII files in a model, open the **.eomodeld** directory. This displays the individual entity, stored procedure, and fetch specification files in the model. The entity files have a **.plist** extension, indicating that the files' contents are in ASCII property list format. The stored procedure files have the extension **.storedProcedure**. The fetch specification files have the extension **.fspec**.You can view the individual files in a text editor.

The global file has the name **index.eomodeld**. It contains the connection dictionary, the adaptor name, and a list of all of the entities in the model.

Models describe the database-to-enterprise object mapping by using the modeling classes EOModel, EOEntity, EOAttribute, and EORelationship (EORelationships include additional information in the form of EOJoin objects).

The following table describes the database-to-object mapping provided in a model:

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

While the modeling classes correspond to elements in a relational database, a model represents a level of abstraction above the database. Consequently, the mapping between modeling classes and database components doesn't have to be one-to-one. So, for example, while an EOEntity object described in a model corresponds to a database table, in reality it can contain references to multiple tables. In that sense, a model is actually more analogous to a database view. Similarly, an EOAttribute can either correspond directly to a column in the root entity, or it can be derived or flattened. A derived attribute typically has no corresponding database column, while a flattened attribute is added to one entity from another entity. For more information, see the chapter "Adding Derived Properties" on page 61.

You can store your model files anywhere, but to use a model in an application you must copy it into your application's project directory.

Starting EOModeler

To start EOModeler on Windows NT, launch EOModeler from the WebObjects program group in the Start menu. On Rhapsody, locate EOModeler in **/System/Developer/Applications** and double-click its icon (shown in Figure 1).

Figure 1. EOModeler's Application Icon

Alternatively, you can start EOModeler by double-clicking an existing model file—from inside the Project Builder application or from the file system.

Tip: On Windows NT, you open a model's **.eomodeld** directory and double-clicking the **index.eomodeld** file to open it in EOModeler.

Creating a New Model

To create a model, choose Model ▶ New.

EOModeler starts the New Model Wizard, which assists you to configure your new model. The following sections describe the wizard pages that guide you through the model creation process.

Selecting an Adaptor

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

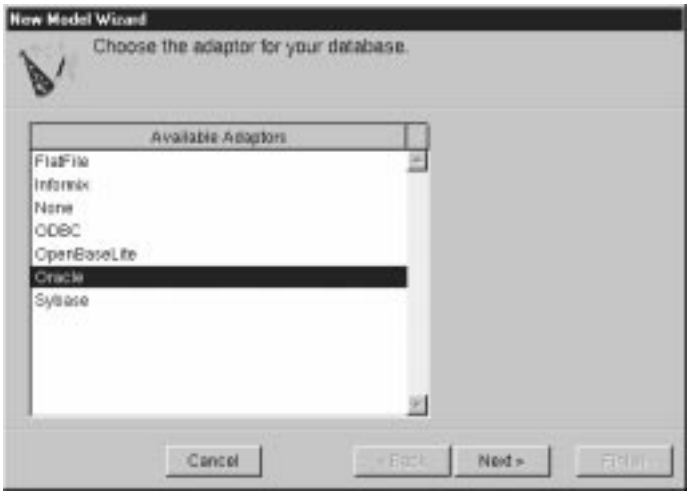


Figure 2. Selecting an Adaptor

After you select an adaptor, EOModeler displays the login panel for the database that corresponds to the adaptor you selected. Fill in the login panel and click OK.

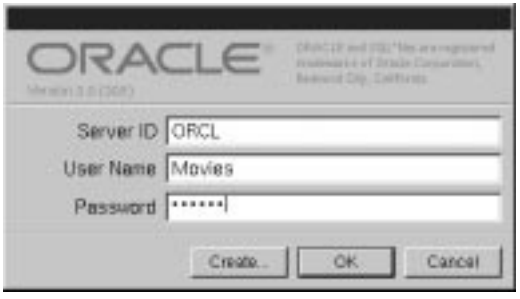


Figure 3. Oracle Login Panel

Different databases require different login information, so each database's login panel looks different.The examples in this chapter use the Oracle version of the Movies database included with the Enterprise Objects Framework; Figure 3 shows the Oracle login panel.

Choosing What to Include in Your Model

In this next wizard page, you can specify the degree to which the wizard configures your model.

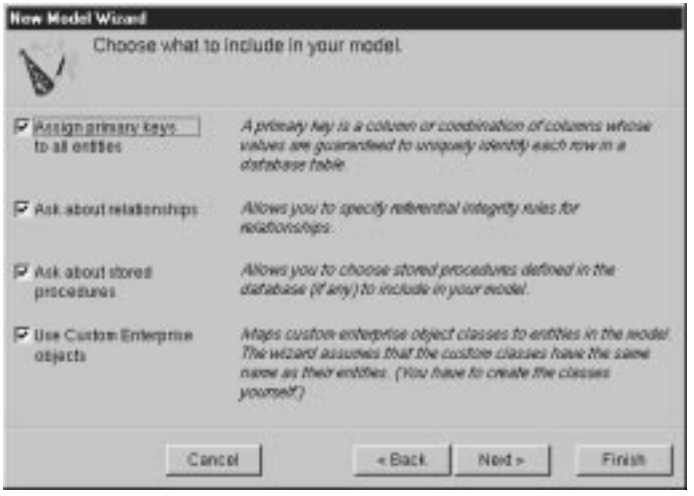


Figure 4. Choosing What to Include in the Model

How complete the model EOModeler creates is depends on how completely the schema information is inside your database server. For example, the wizard includes relationships in your model only if the server's schema information specifies foreign key definitions.

Using the options in this page, you can tell EOModeler that you want to supplement the model with additional information. (Note that the wizard doesn't modify the underlying database.)

Assign primary keys to all entities

Enterprise Objects Framework uses primary keys to uniquely identify enterprise objects and to map them to the appropriate database row. Therefore, you must assign a primary key to each entity you use in your application. The wizard automatically assigns primary keys to the model if it finds primary key information in the database's schema information.

Checking this box causes the wizard to prompt you to choose primary keys that aren't defined in the database's schema information. If your database doesn't define them, the wizard later prompts you to choose primary keys.

Ask about relationships

If there are foreign key definitions in the database's schema information, the wizard includes the corresponding relationships in the model. However, a definition in the schema information doesn't provide enough information for the wizard to set all of a relationship's options. Checking this box causes the wizard to prompt you to provide the additional information it needs to complete the relationship configurations.

Ask about stored procedures

Checking this box causes the wizard to read stored procedures from the database's schema information, display them, and allow you to choose which to include in your model.

Use custom enterprise objects

An entity maps a table to enterprise objects by storing the name of a database table (MOVIE, for example) and the name of the corresponding enterprise object class (a Java class, Movie, for example). When deciding what class to map a table to, you have two choices: EOGenericRecord or a custom class. EOGenericRecord is a class whose instances store key-value pairs that correspond to an entity's properties and the data associated with each property.

If you don't check the "Use custom enterprise objects" box, the wizard maps all your database tables to EOGenericRecord. If you do check this box, the wizard maps all your database tables to custom classes. The wizard assumes that each entity is to be represented by a custom class with the same name. For example, a table named MOVIE has an entity named Movie, whose corresponding custom class is also named Movie.

Use a custom enterprise object class only when you need to add business logic; otherwise use EOGenericRecord.

Choosing the Tables to Include

After specifying what additional information to include in your model, the wizard prompts you to choose the tables to include in your model. By default, all the tables are selected.

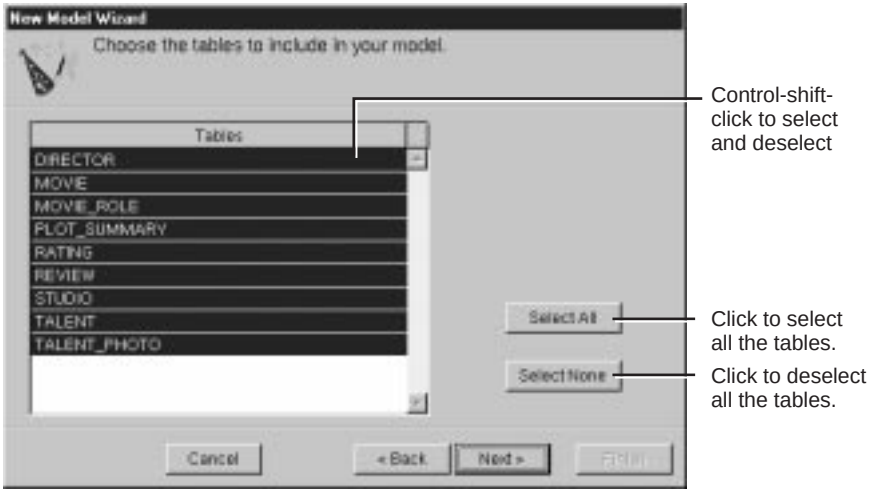


Figure 5. Choosing the Tables to Include

The wizard creates entities only for the tables you select. If you later decide you want to include a table you didn't select at this stage, you can add it using the Model ▶ New Updated Model command, as described in "Updating Your Model" on page 24.

Specifying Primary Keys

If you are using a database that stores primary key information in its database server's schema information, the wizard skips this step. The wizard has already successfully read primary key information from the schema information and assigned primary keys to your model.

However, if primary key information isn't specified in your database server's schema information or if the adaptor can't read it (as with Microsoft Access), the wizard now asks you to specify a primary key for each entity.

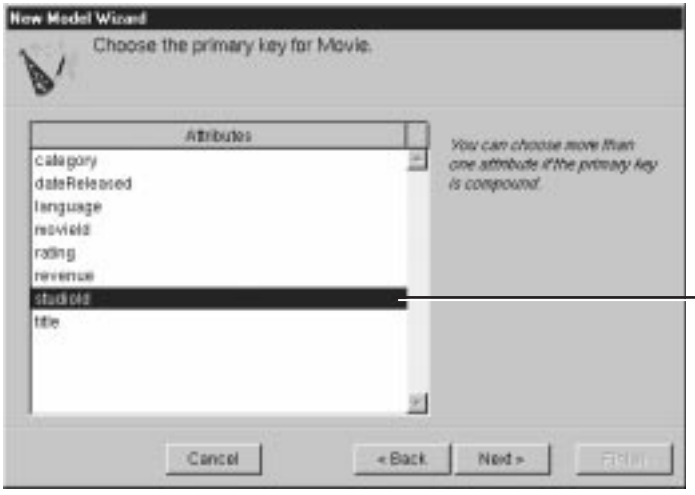


Figure 6. Specifying an Entity's Primary Key

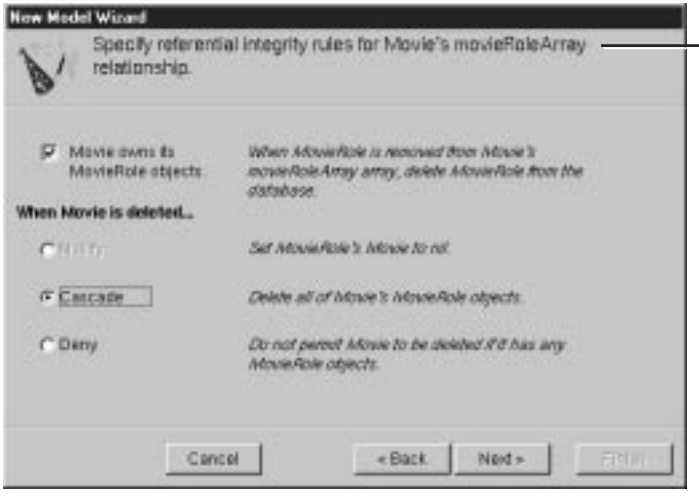
If an entity's primary key is *compound*; that is, if it's composed of more than one attribute, control-shift-click to select all of the attributes in the primary key. You use a compound primary key when any single attribute isn't sufficient to uniquely identify a row. For example, in the MovieRole entity, if requires the combination of the **movieId** and **talentId** attributes to uniquely identify a row.

Specifying Referential Integrity Rules

If foreign key definitions aren't specified in your database server's schema information or the adaptor can't read that information (as with Microsoft Access), the wizard hasn't created any relationships at all, and it skips this step. You can add relationships later as described in the chapter "Working with Relationships" on page 49.

On the other hand, if you're using a database that stores foreign key definitions in its database server's schema information, the wizard reads them and creates corresponding relationships in your model. For example, Movie has a to-many relationship to MovieRole (that is, a Movie has an array of MovieRoles), and Talent has a to-many relationship to MovieRole.

At this point, if you specified that the wizard ask about relationships (as described in "Choosing What to Include in Your Model" on page 17), the wizard now asks you to provide additional information about the relationships so it can further configure them.



The name assigned to the relationship is dependent on the adaptor you're using.

Figure 7. Specifying Referential Integrity Rules for a Relationship

Owns Destination

The checkbox in this page lets you specify whether the relationship's source *owns* its destination objects. When a source object owns its destination objects and you remove a destination object from the source object's relationship array, this also has the effect of 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, in the relationship shown in Figure 7, Movie owns its MovieRole objects.This means that a destination object (MovieRole) can't exist without its source (a Movie). Consequently, when a MovieRole is removed from its Movie's array of MovieRoles, the MovieRole is deleted—deleted in memory and deleted in the database.

Delete Rule

The radio button you choose in this page specifies what to do when the relationship source is deleted: nullify, cascade, or delete. For example, in the relationship shown in Figure 7, when a user tried to delete a Movie, you could:

- Delete the Movie and set all it's MovieRoles not to be associated with a Movie (nullify).
- Delete the Movie and all its MovieRoles (cascade).
- Refuse the deletion if the Movie has MovieRoles (deny).

Choosing Stored Procedures

If you asked the wizard to include stored procedures in your model (as described in "Choosing What to Include in Your Model" on page 17), the wizard asks you to specify which stored procedures to include. By default, all the stored procedures are selected.

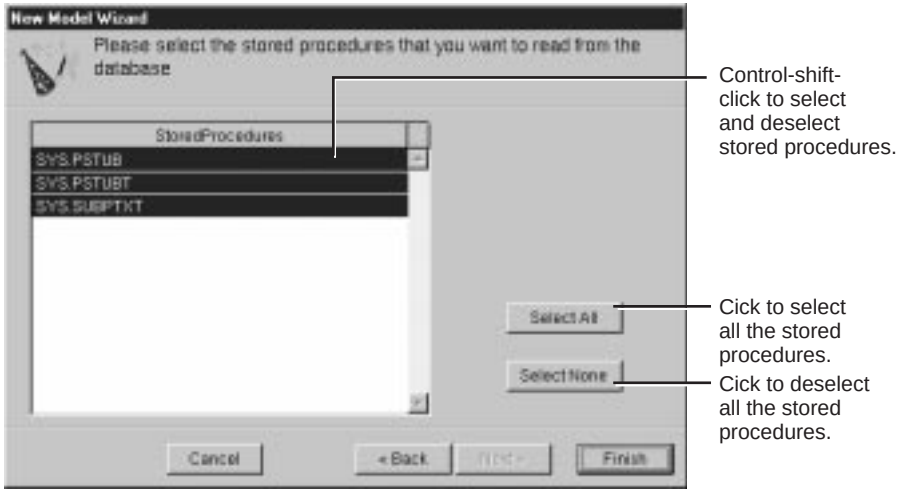


Figure 8. Choosing the Stored Procedures to Include

The wizard includes stored procedure information only for the stored procedures you select. If you later decide you want to include a stored procedure you didn't select at this stage, you can add it using the Property ▶ Add Stored Procedure command, as described in the chapter "Working with Stored Procedures" on page 91.

Saving the Model

When you finish the wizard, EOModeler displays the new model. If you're planning to use your model in an application for which you've already created a project, you should save the model into your project folder. You will be prompted to add it to the project; click OK.

What a New Model Includes

When you create a new model, the information it includes depends on how completely you've specified the underlying database. EOModeler can read all of the following from a database and include it in a default model:

- Table and column names
- Column data types, including the width constraint of string data types
- Primary keys
- User constraints, such as null constraints and uniqueness
- Foreign key definitions (which are expressed in a model as relationships)
- Stored procedures

A model contains not only the information it reads from the database, but values it derives from that information, including:

- Entity and attribute names
- A mapping between the data type of a database column and a corresponding value class, such as String, Number, or NSGregorianDate (NSString, NSNumber, or NSCalendarDate in Objective-C). See the class specification for each adaptor for a listing of the adaptor's default database type to value class mapping.

EOModeler derives entity names by taking a database table name and making all of it lowercase except for the first letter. It then removes underbar (_) characters and capitalizes any characters following underbars. For example:

Database Table	Entity Name
EMPLOYEE	Employee
EMPLOYEE_PHOTO	EmployeePhoto
TEST_OF_SEVERAL_WORDS	TestOfSeveralWords

Attribute names are based on corresponding database columns. They're derived in the same way as entities, except that EOModeler doesn't capitalize the first character. For example:

Database Column	Attribute Name
NAME	name
FIRST_NAME	firstName
MOVIE_ID	movieId

Updating Your Model

When you create a new model in EOModeler, the New Model Wizard prompts you to select the tables you want to include, as described in "Choosing the Tables to Include" on page 19. But what if you decide at a later point that you want your model to include tables you didn't select when you first created it? Or what if your database has been modified, and you want your model to reflect the changes?

To update an existing model, choose Model ▶ New Updated Model. This creates a new model that you can use for cutting and pasting from. Using the New Updated Model command doesn't have a destructive effect on your original model—it just gives you a second model to use for "spare parts."

When you invoke New Updated Model, EOModeler opens a Select Tables panel (shown in Figure 9) that lets you specify the tables you want in the "spare parts" model. By default, the Select Tables panel selects only tables that aren't represented in your working model; accepting the selection creates a new, complementary model.

Figure 9. Selecting Tables for the New Updated Model

Checking for Consistency

EOModeler provides consistency checking to confirm that your model is valid. For example, a model that has entities without primary keys or relationships without join components is not valid.

You can explicitly check your model at any point by choosing Model ▶ Check Consistency. Consistency checking is also invoked automatically whenever you save your model. When a consistency check occurs and inconsistencies are found, the Consistency Check panel appears with a list of diagnostic messages, as shown in Figure 10.

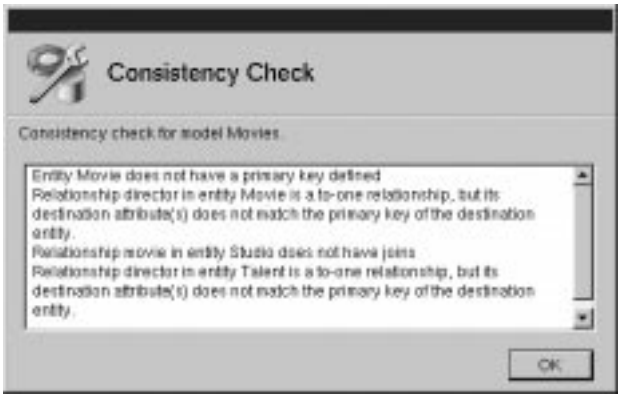


Figure 10. Checking for Consistency

If you prefer for EOModeler not to run the consistency check when you save, you can turn off this behavior with the Preferences panel.

Chapter 2

Using the Model Editor

This chapter describes the *Model Editor*—the window in which EOModeler displays models and in which you view and edit them. The chapter is organized into the following sections:

The Model Editor in Table Mode

EOModeler displays models in the *Model Editor*—the main window in EOModeler. By default, EOModeler uses the *table mode*, as shown in Figure 11. In this mode, the Model Editor has a *tree view* for navigating a model, and a table for editing the model's components (entities, attributes, relationships, and so on).

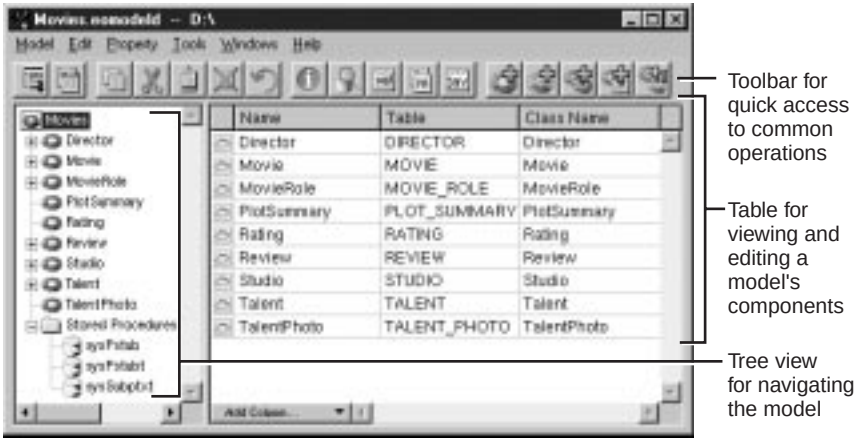


Figure 11. The Model Editor

There are two other modes you can use: the *browser mode* and the *diagram view*. For more information on these, see the section "Using Other Display Modes" on page 34.

Navigating a Model With the Tree View

You navigate a model by clicking icons in the Model Editor's tree view. In Figure 12, the icon labeled Movies (in the upper left corner of the tree view) represents the model itself. You double-click this icon to expand and contract the tree view. When the tree view is expanded, it shows the model's entities.

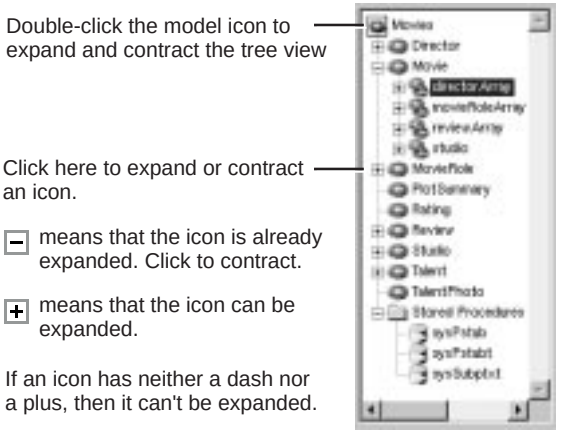


Figure 12. Expanding the Tree View

Similarly, you can expand a model's entities and stored procedures folder. As shown in Figure 12, expanding an entity displays the entity's relationships. A relationship in the tree view represents the relationship's destination entity. Expanding the relationship in the tree view in displays the destination entity's relationships, and so on. Expanding the stored procedures folder displays the model's stored procedures.

You control what's displayed in the Model Editor's table by selecting icons in the tree view. When the model is selected (as shown in Figure 13), the Model Editor displays the model's entities in the table.

To display an entity's attributes and relationships in the table, select the entity. Similarly, to display a stored procedure's attributes, select it.

Figure 13. Changing the Table's Contents

You can also use the icons in the tree view in drag and drop operations—for example, to drag an entity into the Data Browser (described in the chapter "Interacting with a Database" on page 117) or into WebObjects Builder (described in the chapter "Creating a WebObjects Database Application" in the book *Getting Started with WebObjects*).

Displaying a Model's Components in the Table Mode

The Model Editor's table changes depending on what's selected in the tree view. When the model itself is selected, the table displays the model's entities, one entity per row. The columns of the table display information about the entities—entity name, name of the corresponding database table, and so on.

When an entity is selected, the display changes to show two tables: one for the entity's attributes and one for the entity's relationships (shown in Figure 14).

Figure 14. Displaying an Entity's Attributes and Relationships

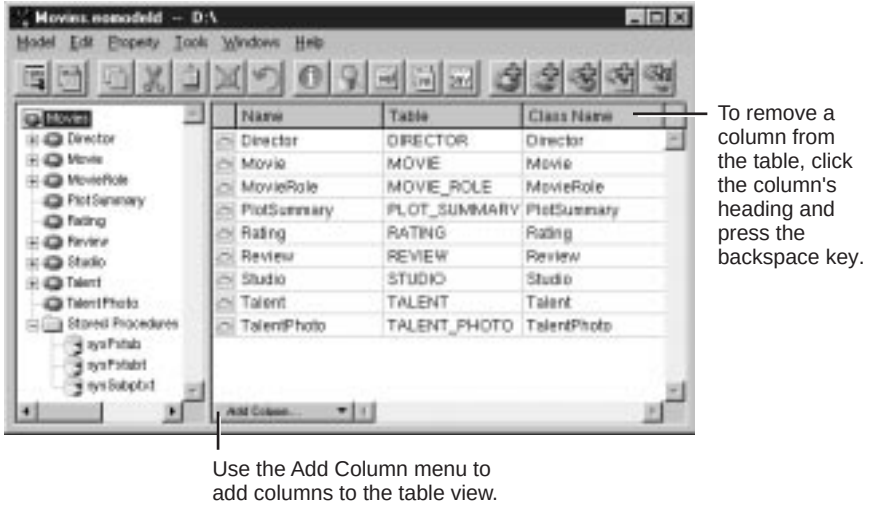
The Open Entity Icon

When the model is selected in the tree view and the table is displaying the model's entities, the Model Editor displays an  icon to the left each entity in the table. Double-clicking this icon *opens* that entity, selecting that entity and displaying its attributes and relationships in the table. You can accomplish the same thing by selecting the entity in the tree view.

Figure 15. Navigating from the Table View

Adding Columns with the Add Column Menu

You use the Add Column menu (shown in) to add columns to the table view. The items in the menu depend on what modeling component the table is displaying and on what columns the table contains. As you add columns to the table, the corresponding menu items are removed from the Add Column menu.



Using Other Display Modes

In addition to the table mode, the Model Editor has two other display modes: browser mode and diagram view. You can switch between display modes with the Change Display View pop-up button on the toolbar as shown in Figure 16.

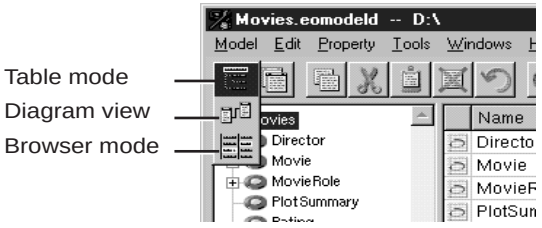


Figure 16. Changing the Model Editor's Display Mode

Diagram View

The diagram view of the Model Editor is very useful for displaying and printing your model graphically. As with the table mode, you can use the diagram view to edit components of the model, but it's editing capabilities are more limited.

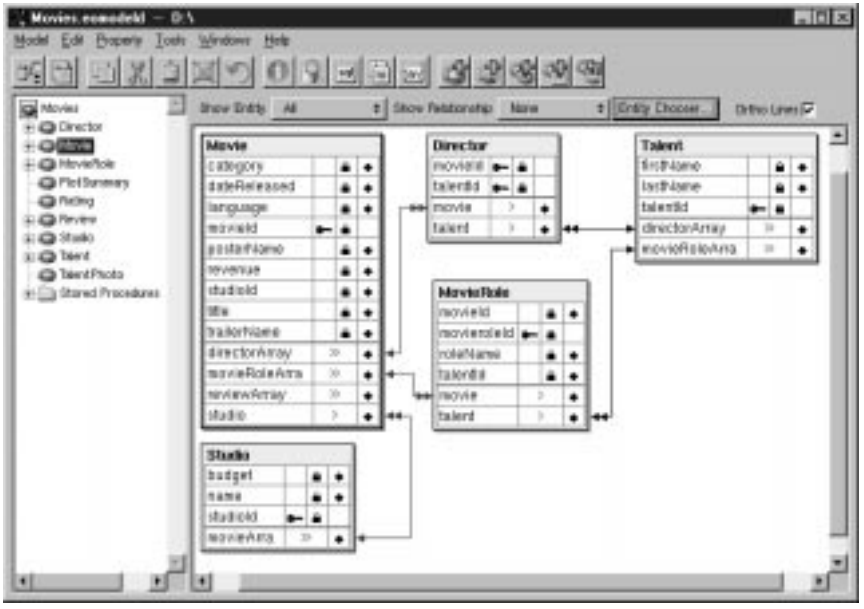


Figure 17. Model Editor Displaying Diagram View

Use the Show Entity and Show Relationship pop-up lists just under the toolbar to specify the information you want to include in the diagram. Click the Entity Chooser button to display the Entity Chooser panel, where you can choose the entities you want to include in the diagram.

Browser Mode

To display the attributes for a particular entity, such as Movie, select the entity. The attributes appear in the column to the right of the entity along with the entity's relationships. You can traverse the model by clicking relationships, as shown in Figure 18

Figure 18. Model Editor in Browser Mode

Chapter 3

Working with Attributes

This chapter describes setting an attribute's characteristics. It's organized in the following sections:

- "Changing an Attribute's Characteristics" (page 39) describes attribute characteristics and the ways you can edit them: using the table mode of the Model Editor, using the diagram view, and using attribute inspectors.
- "Prototype Attributes" (page 45) describes how to assign a prototype to an attribute, how to define your own prototype attributes, and how to hide the prototypes provided by the Framework.

Changing an Attribute's Characteristics

EOModeler provides three mechanisms for viewing and modifying an entity's attributes: the table mode of the Model Editor, the diagram view of the Model Editor, and the Attribute Inspector. You can use any of the mechanisms to examine the characteristics of your model's attributes and to make refinements. Each has advantages over the others and is useful in different circumstances.

- The Model Editor in table mode is most convenient for most attribute editing, because you have access to all of but a few of the possible attribute characteristics.
- Use the diagram view of the Model Editor for very limited attribute editing when you're already using the diagram view. Because you can modify only a few of an attribute's characteristics in diagram view, you have to switch to another mechanism for many tasks.
- Use the Attribute Inspector for editing attribute characteristics that you can't access in the Model Editor's table mode. Additionally, some attribute characteristics are easier to set in the inspector.

The following sections provide more detail on what attribute characteristics you can set with each mechanism.

Using Table Mode

To display an entity's attributes in table mode, select the entity in the tree view.

Figure 19. Displaying an Entity's Attributes

Each table column corresponds to a single characteristic of the attribute, such as its name or its external type (that is, the type by which it's represented in the database). By default, the columns included in this view only represent a subset of the possible characteristics you can set for a given attribute. To add columns for additional characteristics, you use the Add Column menu in the lower left corner of the table.

The following table describes the characteristics you can set for an attribute. Unless otherwise specified, the instructions are for editing the characteristic in the Model Editor's table mode.

Characteristic	What it is	How you modify it
Allows Null	Indicates whether the attribute can have a NULL value.	Click in the column to toggle the check on and off. (The column for Allows Null is labeled "A".)

Class Property	Indicates a property that meets both of these criteria: you want to include it in your class definition, and it can be fetched from the database.	Click in the  column to toggle class property off and on. You can also edit this characteristic in diagram view.
Client-Side Class Property	Declares whether a property is a class property in the entity's client-side class. Only applicable for Java Client applications.	Click in the  column to toggle locking off and on.
Column	The database name of the column that corresponds to the attribute.	Edit the table cell.
Definition	The definition for a derived attribute. Note that Column and Definition are mutually exclusive; you can't set both. Setting one clears the other.	Edit the table cell.
External Type	The data type of the attribute as it's understood by the database.	Choose a value from the pull-down list
Locking	Indicates whether an attribute should be used for locking when an update is performed.	Click in the  column (shown in Figure 19) to toggle locking off and on. You can also edit this characteristic in diagram view.
Name	The name your application uses for the attribute. EOModeler supplies default names derived from the name of the corresponding column in the database. You can edit these names if desired.	Edit the table cell. You can also edit this characteristic in diagram view.
Precision	The number of significant digits.	Edit the table cell. Or use the Attribute Inspector as described in "Using the Attribute Inspector" on page 43.
Primary Key	Declares whether a property is, or is part of, the primary key for the entity.	Click in the  column to toggle the primary key off and on. You can also edit this characteristic in diagram view.
Prototype	A prototype attribute from which this attribute derives its characteristics.	Choose a value from the pull-down list

Read Format	The format string that's used to format the attribute's value for SELECT statements. In the string, %P is replaced by the attribute's external name. This string is used whenever the attribute is referenced in a select list or qualifier.	Edit the table cell.
Read Only	Indicates whether the attribute is read only.	Use the Advanced Attribute Inspector. You can't set this characteristic in the Model Editor.
Scale	The number of digits to the right of the decimal point. Can be negative.	Edit the table cell. Or use the Attribute Inspector as described in "Using the Attribute Inspector" on page 43.
Value Class (Java)	If your enterprise object is a Java class, the Java type to which the attribute will be coerced in your application.	Edit the table cell. Or use the Attribute Inspector as described in "Using the Attribute Inspector" on page 43.
Value Class (Obj-C)	If your enterprise object is an Objective-C class, the Objective-C type to which the attribute will be coerced in your application.	Edit the table cell. Or use the Attribute Inspector as described in "Using the Attribute Inspector" on page 43.
Value Type	The conversion character (such as "i" or "d") for the data type a NSNumber attribute is converted to and from in your application.	Edit the table cell. Or use the Attribute Inspector as described in "Using the Attribute Inspector" on page 43.
Width	The maximum width (applies to string and raw data only).	Edit the table cell. Or use the Attribute Inspector as described in "Using the Attribute Inspector" on page 43.
Write Format	The format string that's used to format the attribute's value for INSERT or UPDATE expressions. In the string, %P is replaced by the attribute's external name.	Edit the table cell.

Using the Attribute Inspector

The Attribute Inspector is most useful for setting characteristics of an attribute that are related to how the attribute is represented inside your application. These characteristics are:

- Precision
- Scale
- Value Class (Java)
- Value Class (Obj-C)
- Value Type
- Width

The Attribute Inspector is particularly useful for editing these characteristics because it manages the dependencies between them. For example, EOModeler supplies a default mapping between an attribute's external type and internal type, both for Java and Objective-C. If you change the Java value class, the Objective-C value class is automatically updated correspondingly, and vice versa. Also, some of the attribute characteristics are only applicable to attributes whose internal type is set to a particular value class. For example, Precision and Scale are only applicable to decimal number value classes—BigDecimal in Java and NSDecimalNumber in Objective-C.

The Attribute Inspector helps you keep track of these dependencies by changing its user interface to match whatever internal data type you choose (shown in Figure 20).

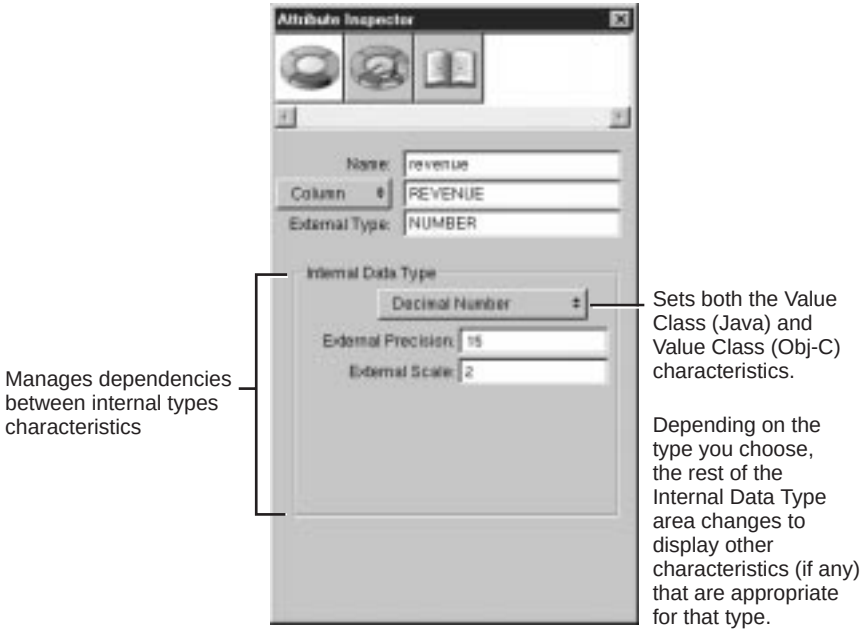


Figure 20. Setting Internal Data Type Characteristics with he Attribute Inspector

To view an attribute in the Attribute Inspector, select the attribute in the Model Editor and open the inspector, either by clicking the button in the toolbar or by choosing Tools ▶ Inspector.

Using Custom Data Types

Some attributes, such as TalentPhoto's **photo** attribute, use custom value classes to represent them inside your application. When you use a custom data type, you are responsible for specifying how the data is read from and written to the database. You can use the Attribute Inspector to specify a custom data type. For a description of how to do this, see the chapter "Advanced Enterprise Object Modeling" in the book *Enterprise Objects Framework Developer's Guide*. See the class specification for EOAttribute in the *Enterprise Objects Framework Reference* for more discussion of custom data types.

Using the Advanced Attribute Inspector

The main reason you use the Advanced Attribute Inspector is to set an attribute's Read Only characteristic. By default, and attribute is read-write. You only need to set it if you want it to be read-only. To do this, you have to use the Advanced Attribute Inspector. Open the inspector panel, and select the Advanced Attribute Inspector as shown in Figure 21.

Figure 21. Displaying the Advanced Attribute Inspector

Prototype Attributes

To allow easier model creation from scratch, EOModeler supports the concept of prototype attributes. Prototype attributes are just what they sound like — special attributes from which other attributes derive their settings. A prototype can specify any of the characteristics you normally

define for an attribute. When you create an attribute, you can associate it with one of these prototypes, and the attribute's characteristics are then set from the prototype definition.

For example, suppose your adaptor contains a date prototype that defines the value class to be NSGregorianDate (NSCalendarDate in Objective-C) and the external type to be DATE. When you create an attribute and associate it with this date prototype, the attribute's value class is dynamically resolved to NSGregorianDate and its external type is dynamically resolved to DATE.

Assigning a Prototype to an Attribute

To associate an attribute with a prototype, use the table mode of the Model Editor. Simply choose a prototype from the combo box in the Prototype column as shown in Figure 22. If EOModeler isn't displaying the Prototype column, activate it from the Add Column menu.

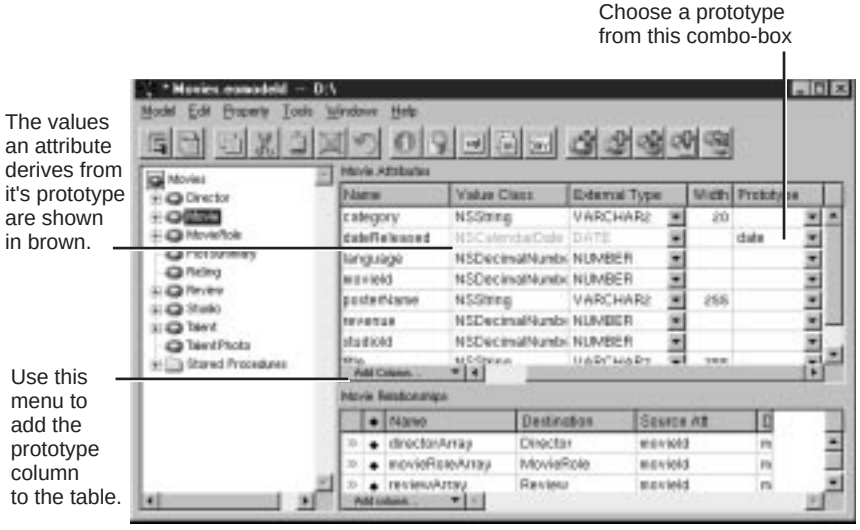


Figure 22. Assigning a Prototype to an Attribute

If any of the prototype information is incorrect for your attribute, you can override it. Just set the property of the attribute to the value you want (see Figure 23). The remaining attribute properties will still dynamically resolve to the values set in the prototype.

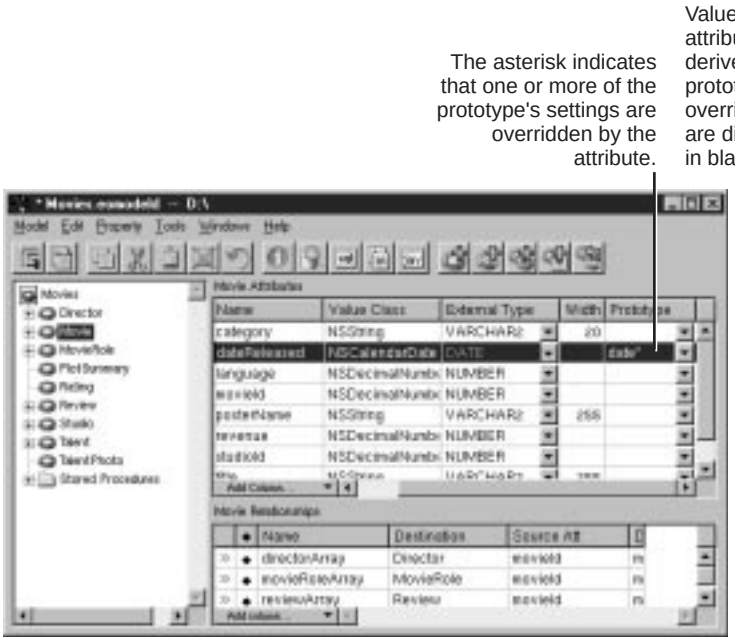


Figure 23. Overriding Prototype Settings

Creating Prototype Attributes

The prototypes you can assign to an attribute come from three places:

1. An entity named **EO<AdaptorName>Prototypes**, where **<AdaptorName>** is the name of the adaptor for your model (EOOraclePrototypes, for example)

2. An entity named **EOPrototypes**

3. The adaptor for your model

So to create your own prototype, create a prototype entity—an entity named either **EO<AdaptorName>Prototypes** or **EOPrototypes**—and add an attribute to it. Note that the **EO<AdaptorName>Prototypes** and **EOPrototypes** entities can be defined in the current model or in another model in the model group (all the models in your project are typically a part of the same model group).

When resolving a prototype name, Enterprise Objects Framework looks for prototypes in **EO<AdaptorName>Prototypes**, then in **EOPrototypes**, and finally in the adaptor for your model. This search path allows you to override the prototypes provided by each adaptor. Furthermore, if you don't want to use the adaptor-defined prototypes at all, you can hide them. Create an entity named **EOPrototypesToHide**. For each prototype you want to hide, create an attribute with that name; you don't need to specify other attribute properties.

Chapter 4

Working with Relationships

This chapter describes creating and configuring relationships. It's organized in the following sections:

Creating Relationships

If the database on which your model is based includes definitions for foreign keys, these definitions will automatically be expressed in your model as ready-made relationships.

You can also explicitly form a relationship between entities if one doesn't already exist. This relationship must reflect an actual relationship between the entities' corresponding tables in the database.

Forming a relationship allows you to access data in a destination table that relates to data in a source table (it's also possible to have a reflexive relationship, in which the source and destination tables are the same). For example, to find all of the roles in a particular movie, you can form a relationship between the MovieRole and Movie entities.

EOModeler provides two mechanisms for forming relationships. You can form them in the Model Editor's diagram view or in the Relationship Inspector. Using the diagram is the quickest way to create a new relationship, but using the Relationship Inspector gives you access to more relationship characteristics. Each mechanism is discussed in the following sections.

Forming Relationships in the Diagram View

To create a relationship in diagram view, control-drag from a source attribute to the destination attribute, as shown in Figure 24.

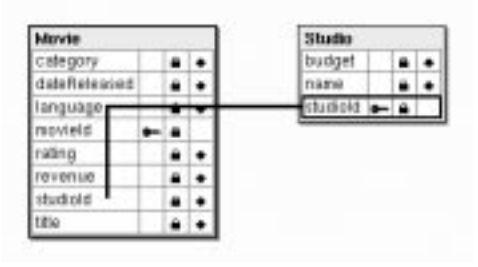


Figure 24. Control-Dragging to Create a Relationship

Control-dragging to create a relationship actually creates two relationships: one in the source attribute's entity and an inverse relationship in the destination attribute's entity. So in Figure 24, control-dragging from the Movie entity's **studioId** attribute to the Studio entity's **studioId** attribute creates the relationships:

- **studio**, a to-one relationship in Movie to Studio
- **movies**, a to-many relationship in Studio to Movie

You can view the new relationships in the Model Editor's table mode and you can further configure them in the Relationship Inspectors as described in the next sections.

Forming Relationships in the Relationship Inspector

Creating a relationship with the Relationship Inspector is a more manual process than creating one in the diagram view. The inspector provides only the ability to configure a relationship that already exists. Consequently, unlike with the diagram view, you have to create a relationship before you can edit it with the Relationship Inspector.

1. Select a source entity in the Model Editor, such as Movie.
2. Choose Property ▶ Add Relationship.

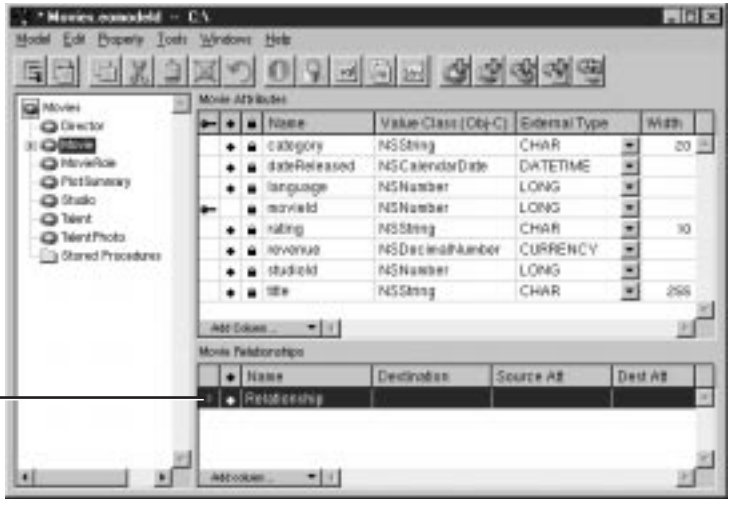


Figure 25. Adding a Relationship

Alternatively, you can click the button in the toolbar. In either case, the text "Relationship" appears in the relationship table at the bottom of the window.

3. Select the new relationship in the Model Editor.
4. Open the Relationship Inspector, either from the toolbar or by choosing Tools ▶ Inspector.

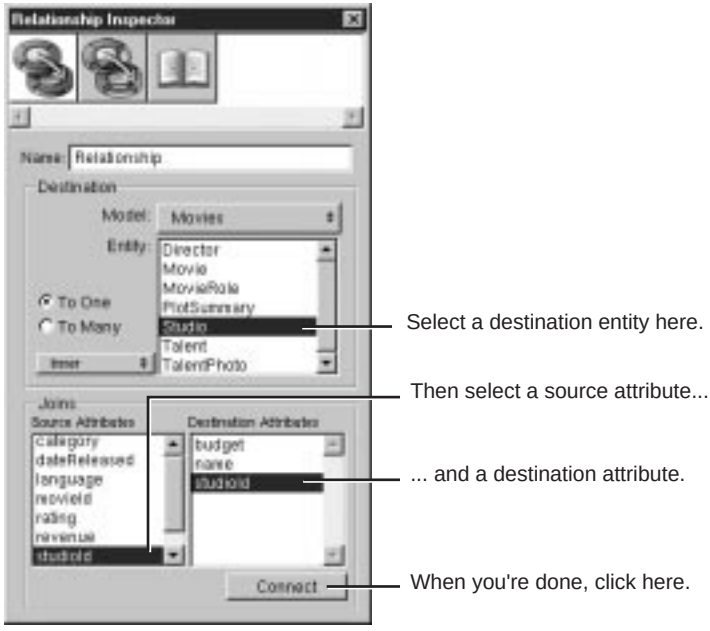


Figure 26. The Relationship Inspector

5. In the Inspector, select the destination entity (Studio) in the Destination browser.

   Typically, you form a relationship by connecting a primary key in one entity and a corresponding foreign key in another entity. In a to-one relationship, the source entity usually holds the foreign key, while the destination entity holds the primary key. The opposite is true for a to-many relationship. For example, **studioId** is a foreign key for Movie, while it's the primary key for Studio.

6. Select the source attribute (**studioId**) in the Source Attributes browser.

7. Select the destination attribute (**studioId**) in the Destination Attributes browser

8. Make sure the relationship has the proper cardinality (in this example it should be set to To One since a movie has only one Studio).

9. Click Connect.

EOModeler assigns the relationship a default name; in this example it's "studio." You can edit this name if desired using either the Inspector or the table view.

Forming Relationships Across Models and Databases

The entities in one model can have relationships to the entities in another model. You can form such relationships even if the models map to different databases and different database servers.

When you add a model to a project, it becomes part of a model group, even if the model group only contains that one model (for more information on model groups, see the EOModelGroup class specification in the *Enterprise Objects Framework Reference*). Each subsequent model that you add to the project—either directly by adding the model to the project's Resources suitcase or indirectly by adding a framework that includes a model—automatically becomes part of the group. Entity names must be unique within a model group; you can't use the same entity name in two different models in the same group. Put another way, all the entities used in an application must have unique names.

To form a relationship from one model to another, use the Relationship Inspector as follows:

1. Add a relationship to the entity you want to use as the source of the relationship.

   For example, you can form a to-one relationship between the Movie entity in the Movies sample database and the VideoTape entity in the Rentals sample database.

2. In the Relationship Inspector, use the Model pop-up list to choose the model containing the entity you want to use as the destination of the relationship.

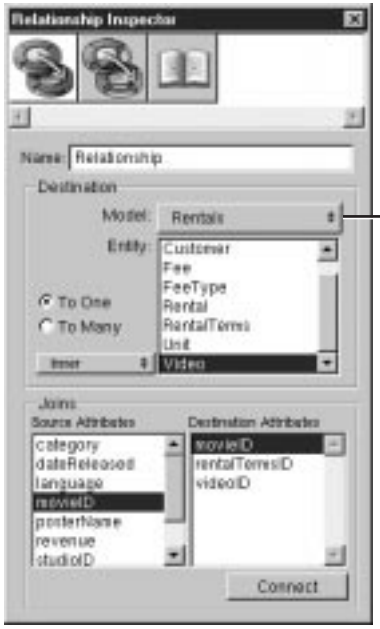


Figure 27. Creating a Relationship Across Models

3. Specify the relationship as you normally would.

Note: You can't flatten properties across databases, nor can you map inheritance hierarchies across databases (though you can do both of these things across models that map to the same database).

Tips for Specifying Relationships

The following tips are useful to keep in mind as you specify relationships in your model:

- The relationships you define in your model must reflect a corresponding implementation in the database, as well as the features supported by your adaptor. EOModeler doesn't know, for example, if a relationship is to-one or to-many, or if your adaptor supports left outer joins. You need to know your database and your adaptor, and specify relationships accordingly.

- Use the diagram view to quickly create pairs of inverse relationships by control-dragging between source and destination attributes.
- Use the Relationship Inspector to specify information about a relationship, such as whether it's to-one or to-many, its semantics (that is, the type of join represented by the relationship), and the name of the destination model (if the destination isn't in the current model).
- Relationships are created as to-one relationships. You need to change this setting if the two entities have a to-many relationship (for example, a movie has many roles).
- A relationship can be compound, meaning that it can consist of multiple pairs of connected attributes. You can specify additional pairs of attributes only in the Relationship Inspector. Simply select a second source attribute and a second destination attribute, and click Connect a second time.
- A to-one relationship from one primary key to another primary key must always have exactly one row in the destination entity—if this isn't guaranteed to be the case, use a to-many relationship. This rule doesn't apply to a foreign key to primary key relationship, where a NULL value for the foreign key in the source row indicates that no row exists in the destination.
- To-one relationships must join on the complete primary key of the destination entity as the join component.

For more discussion about modeling relationships, see the chapter "Advanced Enterprise Object Modeling" in the book *Enterprise Objects Framework Developer's Guide*.

Adding Referential Integrity Rules

You can use the Advanced Relationship Inspector to add referential integrity rules for a relationship.

To add referential integrity rules:

1. Select the relationship for which you want to add rules.
2. In the Relationship Inspector, click the Advanced Relationship Inspector icon as shown in Figure 28.

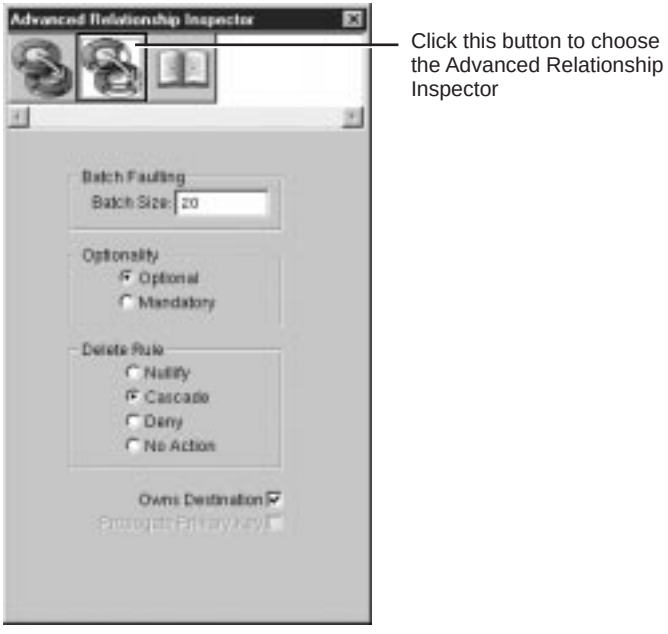


Figure 28. Advanced Relationship Inspector

You can use the fields in the Advanced Relationship Inspector to further specify a relationship. The options in this inspector are described in the following sections.

Batch Faulting

Normally when a fault is triggered, just that object (or array of objects for a to-many relationship) is fetched from the database. You can take advantage of this expensive round trip to the database by batching faults together. The value you type in the Batch Size field indicates the number of faults for the same relationship that should be triggered along with the first fault. For more discussion of batch faulting, see the class specification for EODatabaseContext in the *Enterprise Objects Framework Reference*.

Optionality

This field lets you specify whether a relationship is optional or mandatory. For example, you could require all departments to have a location (mandatory), but not require every employee to have a manager (optional).

Delete Rule

This field lets you specify the delete rules that should be applied to an entity that's involved in a relationship. For example, you could have a department with multiple employees. When a user tried to delete the department, you could:

- Delete the department and remove any back reference the employee has to the department (Nullify).
- Delete the department and all of the employees it contains (Cascade).
- Refuse the deletion if the department contains employees (Deny).
- Allow the deletion and do nothing to the destination objects (No Action).

The No Action rule is useful for tuning performance. However, you should use this delete rule with great caution since it can result in dangling references in your object graph. For more information, see the class specification for EOClassDescription in the *Enterprise Objects Framework Reference*.

Owns Destination

The Owns Destination checkbox lets you 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, this also has the effect of 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.

Propagate Primary Key

The Propagate Primary Key checkbox lets you specify that the primary key of the source entity should be propagated to newly inserted objects in the destination of the relationship. This is typically used for an 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.

Chapter 5

Adding Derived Properties

The Enterprise Objects Framework supports three different kinds of attributes: simple, derived, and flattened. Simple attributes, which correspond to a single column in the root table of the entity and can be read or updated directly from or to the database, are described in the chapter "Working with Attributes" on page 37. This chapter describes the other two types.

Additionally, the Framework supports two kinds of relationships. Simple relationships, which relate one table to another with a join, are described in the chapter "Working with Relationships" on page 49. The other kind, flattened relationships, are described in this chapter.

It's organized into the following sections:

Derived Attributes

A derived attribute doesn't map directly to a single column in the root table of the entity. A derived attribute can be based on another attribute that's modified in some way, such as an **bonus** attribute that's the result of a calculation performed on a **salary** attribute. A derived attribute can also be an aggregate consisting of more than one attribute; for example, you can create a derived attribute **fullName** that is an aggregate of **lastName** and **firstName**.

Derived attributes, since they don't correspond to real values in the database, are read-only; it makes no sense to write a derived value.

Adding a Derived Attribute

You can use the concept of derived attributes to add to an entity a new attribute that doesn't correspond to any database column. This attribute can contain a computed value, for example, or an aggregate of multiple attributes.

To add a new attribute to your entity:

1. In the Model Editor, select the entity (such as Talent) to which you want to add an attribute.

2. Choose Property ▶ Add Attribute.

   Alternatively, you can use the  button on the toolbar. In either case, a new attribute with the name "Attribute" appears in the entity's list of attributes.

3. In the Attribute Inspector, edit the Name field to supply a new name for the attribute.

   For example, you can create an attribute called **fullName** that combines the **firstName** and **lastName** attributes.

   Note that this is a contrived example. A safer way to achieve the same end would be to implement a method on your enterprise object—this would ensure that if the **firstName** or **lastName** attribute is modified, the derived attribute **fullName** will immediately reflect the change.

4. Use the pop-up list to the left of the Definition field to change the attribute type from Column to Derived.

5. Edit the Definition field to supply the SQL needed to specify the derived attribute.

   For example, to concatenate the **firstName** and **lastName** attributes in Oracle, type the text:

   ```
   firstName||' '||lastName
   ```

   The Sybase equivalent is:

   ```
   firstName+' '+lastName
   ```

6. In the External Type field, add the attribute's data type (VARCHAR2). This should be the data type as it is in the database.

7. In the External Width field, type the width constraint for the attribute (this only applies to string and data values).

Figure 29 shows the Attribute Inspector with the new attribute **fullName** specified.

Figure 29. Adding a Derived Attribute

The text you supply in the Definition field must be valid SQL for your database. While you can use either the internal or external names for simple attributes in this field, for derived and flattened attributes you have to use the internal names (flattened and derived attributes have no external names). For consistency's sake, you may want to use only internal names in this field.

Flattened Attributes

A flattened attribute is a special kind of attribute that you effectively add from one entity to another by traversing a relationship. When you form a to-one relationship between two tables (such as MovieRole and Talent), you can add attributes from the destination entity to the source entity—for example, you can add a **lastName** attribute to MovieRole to identify the actor who played a particular role. This is called "flattening" an

attribute. Flattening an attribute is equivalent to creating a joined column; it allows you to create objects that extend across tables.

When Should You Use Flattened Attributes?

Flattening attributes is just one way to conceptually "add" an attribute from one entity to another. A generally better approach is to traverse the object graph directly through relationships. Both WebObjects Builder and Interface Builder make this easy by supporting the notion of *key paths*. You can also access the values in other objects programmatically through relationship references, as described in the chapter "Designing Enterprise Objects" in the book *Enterprise Objects Framework Developer's Guide*.

The difference between flattening attributes and traversing the object graph (either programmatically or by using key paths) is that the values of flattened attributes are tied to the database rather than the object graph. If an enterprise object in the object graph changes (for example, because a user changed a value in another part of the application), a flattened attribute can quickly get out of sync. For example, suppose that you flatten a **deptName** attribute into an Employee object. If a user then changes the employee's department reference to a different department or changes the name of the department itself, the flattened attribute won't reflect the change until the changes in the object graph are committed to the database and the data is refetched. However, if you're using key paths in this scenario, a user of your application sees changes to data as soon as they happen in the object graph. This ensures that your application's view of the data remains internally consistent.

Therefore, you should only use flattened attributes in the following cases:

- If you want to combine multiple tables joined by a one-to-one relationship 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 need to 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 is read-only.

- If you're using vertical inheritance mapping (as described in *Enterprise Objects Framework Developer's Guide*'s chapter "Advanced Enterprise Object Modeling").

Flattening an Attribute

To flatten an attribute, the relationship you traverse must be a to-one relationship.

To flatten an attribute:

1. In the browser mode of the Model Editor, select the relationship that gives you access to the attribute you want to add to your entity (you don't have to use browser mode, it just makes it easier to see the results of the operation).

   For example, to add the name of an actor to MovieRole, you can traverse a **talent** relationship (which represents MovieRole's relationship to Talent) and add the actor's last name (**lastName**) to MovieRole as a flattened attribute. This is a contrived example, because in this case it would be better to use a key path than to flatten an attribute.

2. Select the attribute you want to add (**lastName**), and choose Property ▶ Flatten Property.

Figure 30. Adding a Flattened Attribute

Alternatively, you can use the  button in the toolbar.

The derived attribute (in this example, **talent_lastName**) appears in the list of properties for MovieRole. The format of the name reflects the traversal path: the attribute **lastName** is added to MovieRole by traversing the **talent** relationship.

3. Examine the derived attribute (**talent_lastName**) in the Attribute Inspector.

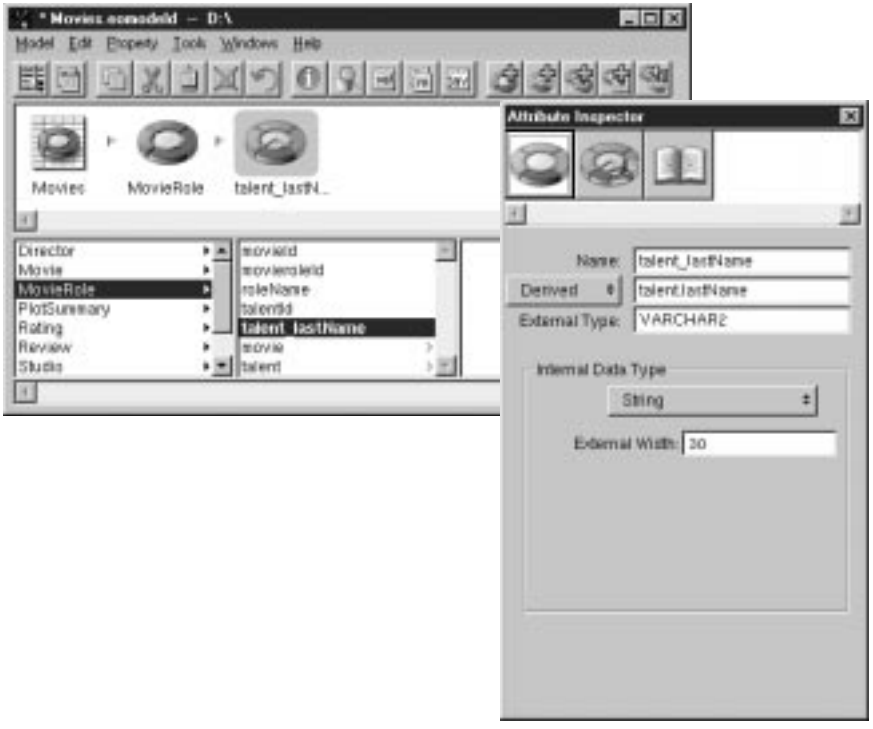


Figure 31. Examining a Flattened Attribute in the Attribute Inspector

In the Attribute Inspector, the pop-up list to the left of the Definition field identifies the attribute as “Derived”.

4. Edit the Name text field to simplify the attribute name (for example, to “lastName”).

The Definition field (the second text field from the top of the Attribute Inspector) must accurately reflect the attribute's external name and the table in which it resides. For example, if you edit its text to be "Name" and change its mode to "Column," it no longer maps to an existing attribute. In other words, only edit this field if you are sure you can predict the outcome.

To display this flattened attribute, use the Data Browser.

5. Select the flattened attribute along with another "native" attribute for verification purposes.

   To select multiple, non-contiguous attributes in the Model Editor on Windows NT, hold down the Control key while you click on each attribute. On Rhapsody, use the Shift key.

6. View the attributes in the Data Browser as shown in Figure 32

Figure 32. Using the Data Browser to Check Your Model

Displaying data associated with your model in the Data Browser is a good way to check that the model is synchronized with the database. If your model is out of sync with the database (for example, if you try to implement a relationship for which there is no corresponding relationship in the database), attempting to display data in the Browser will fail. For more discussion of the Data Browser, see the chapter "Interacting with a Database" on page 117.

Flattened Relationships

In addition to flattening attributes, you can also flatten relationships. Flattening a relationship gives a source entity access to relationships that a destination entity has with other entities. It's equivalent to performing a multi-table join. Note that flattening either an attribute or a relationship can result in degraded performance when the destination objects are accessed, since traversing multiple tables makes fetches slower.

When Should You Use Flattened Relationships?

As discussed in "When Should You Use Flattened Attributes?" on page 66, flattening is a technique you should only use under certain conditions. Instead of flattening an attribute or a relationship, you can instead directly traverse the object graph, either programmatically or by using key paths. This ensures that your application has an internally consistent view of the data.

There is one scenario in which you might want to use a flattened relationship: if you're modeling a many-to-many relationship and you want to perform a multi-table hop to access a table that lies on the other side of an intermediate table. For example, in the Movie database, the Director table acts as an intermediate table between Movie and Talent. It's highly unlikely that you would ever need to fetch instances of Director into your application. In this situation, it makes sense to specify a relationship between Movie and Director, and flatten that relationship to give Movie access to the Talent table.

Flattening a Relationship

To flatten a relationship:

1. Add a relationship from one entity (*entity_1*) to a second entity (*entity_2*).

   For example, you can add a to-many relationship called **toDirectors** from Movie to Director since a movie can have more than one director.

2. Add a relationship from *entity_2* to a third entity (*entity_3*).

   For example, you can add a to-one relationship called **talent** from Director to Talent. For each director a movie has, there is a corresponding single entry in the Talent table.

3. From *entity_1*, select the relationship to *entity_2* to display its properties.

   From Movie, select the relationship **toDirectors** to display the properties of Director.

4. In the list of properties for *entity_2*, select the relationship (**talent**) you want to flatten.

5. Choose Property ▶ Flatten Property.

Figure 33. Flattening a Relationship

The flattened relationship (in this example, **toDirectors_talent**) appears in the list of properties for Movie. The format of the name reflects the traversal path: The relationship **talent** is added to Movie by traversing the **toDirectors** relationship.

Chapter 6

Working with Entities

The work you do with entities falls into two basic categories:

- Using the Model Editor and the Entity Inspectors to set the entity's characteristics.
- Optionally, mapping the entity to a custom enterprise object class and generating source files for it.

This chapter describes these tasks. It's organized into the following sections:

Changing an Entity's Characteristics

To display a model's entities in table mode, select the model (the first icon) in the tree view.

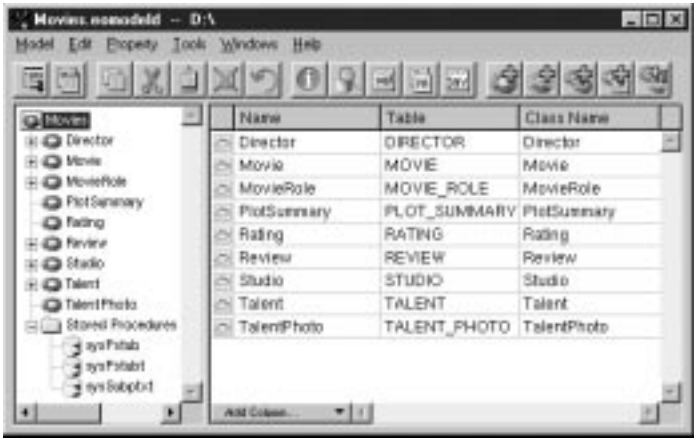


Figure 34. Displaying an Entity's Attributes

Each table column corresponds to a single characteristic of an entity, such as its name or the name of its database table. By default, the columns included in the table—Open Entity, Name, Table, and Class Name—only represent a subset of the possible characteristics you can set for a given entity. To add columns for additional characteristics, you use the Add Column menu in the lower left corner of the table.

The following table describes the characteristics you can set for an entity in the Model Editor.

Characteristic	What it is
Class Name	The name of the class that corresponds to the entity. If you don't define a custom enterprise object class for an entity, by default its class is EOGenericRecord.
Client-Side Class Name	The name of the class that corresponds to the entity in the client-side of a Java Client application. If you don't define a client-side class, the Framework looks for a class in the client with the same name as the server-side enterprise object class. If no such class exists on the client, it uses EOGenericRecord.
External Query	Any SQL statement that will be executed as is—on Sybase, this can be a stored procedure.
Name	The name your application uses for the entity. By default, EOModeler supplies a name based on the name of the corresponding table in the database.
Open Entity	Adds a column with the Open Entity icon, which you can double-click on to display an entity's attributes.
Parent	Indicates an entity's parent—used to model inheritance.
Read Only	Indicates whether the entity is read-only or not.
Table	The name of the database table that corresponds to the entity.

Note: There are numerous other characteristics that you set using the Entity Inspectors

Using the Entity Inspector

You use the Entity Inspector to set an entity's characteristics and specify a mapping between the entity and an enterprise object class. You can also accomplish the same tasks using the table mode of the Model Editor, but this section focuses on the Entity Inspector.

To inspect an entity, select the entity in the Model Editor and open the inspector (either with the  button on the tool bar or by choosing Tools ▶ Inspector).

Figure 35 shows the Entity Inspector for the Movie entity.

Figure 35. The Entity Inspector

Name and Table Name

The Name field lists the name your application uses for the entity. The Table Name field contains the name of the root table in the database. You can change the internal name (that is, the name as it

appears in the application), but you shouldn't change the database table name unless you have also changed the name in your database server.

Class

The Class field initially contains the text "EOGenericRecord". This is because the default enterprise object class is an EOGenericRecord. To specify a custom class, type the name of the class in this field. For more information on creating custom classes, see "Specifying an Enterprise Object Class" on page 79.

Properties

The Properties area lets you specify the properties you want to include in your enterprise object class and set characteristics for them.

There are three columns in this area. Each column displays the status of a particular setting: Primary Key, Used For Locking, and Class Property. Icons are used to indicate that a setting is enabled for a particular property; the dash icon indicates that a setting is not applicable to a property. You add and delete icons by clicking the appropriate column next to the property.

 The Primary Key column is used to declare whether a property is, or is part of, the primary key for the enterprise object class. To specify a compound primary key, you simply add a Primary Key icon to the column for each property you want to include in the primary key.

Specifying a primary key for your enterprise object class is mandatory; the primary key is the means by which an enterprise object is uniquely identified within your application and mapped to the appropriate database row.

Note: Enterprise Objects Framework doesn't support modifiable primary key values. You shouldn't design your application so that users can change a primary key's value.

 The Class Property column is used to indicate properties that meet both of these criteria: You want to include them in your class definition, and they can be fetched from the database. By default, the Entity Inspector sets all of an entity's properties as belonging to your class. You can remove a property by clicking its Class Property icon. If you define an

attribute that doesn't exist in the database but is used by your application (such as a computed value), you should remove its Class Property icon. Note that generated source files won't include instance variable declarations for these attributes—you'll have to type those in by hand (this is a rare case). You also should not include primary and foreign keys as class properties unless you need to display their values in the user interface. If you don't remove the Class Property icon for an attribute that has no corresponding database value, it will result in a server error when your application attempts to fetch the property from the database.

 The Used For Locking column 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 have an adverse effect on system performance. By default, the Entity Inspector sets all of an entity's attributes to be used for locking.

In Figure 35, note that:

- In the Inspector, the property **movieID** has been designated as the enterprise object class's primary key.
- For the entity's relationships, the Inspector automatically displays the Not applicable icons in the Primary Key and Used For Locking columns.

Specifying an Enterprise Object Class

Specifying an enterprise object class for an entity applies the mapping defined in your model to your custom class, thereby enabling objects of the class to be created from database rows.

To specify the enterprise object class for an entity:

1. Make sure that each of the properties you want to include in your enterprise object class has a Class Property icon set for it in the Inspector.

2. If the entity does not already have a primary key specified, add a Primary Key icon for the property or properties that constitute the entity's primary key.

   Remember that the primary key or keys you set for your enterprise object class must mirror the primary key or keys defined for the corresponding table in the database.

What you do after this point depends on how you plan to implement your enterprise object class. In all cases, an enterprise object class must conform to the EOKeyValueCoding interface (or informal protocol in Objective-C), which specifies methods for accessing values by name, or *key* ("keys" in this context relates to key-value pairs, not to primary keys). But this can be accomplished very differently, depending on the approach you use.

You can use either of the following approaches, depending on the needs of your application:

- Use EOGenericRecord.

  If you don't edit the Class field to specify a name for a custom class, the Framework uses EOGenericRecord as an enterprise object class by default. A generic record uses a dictionary to store key-value pairs that correspond to an entity's properties and the data associated with each property. Use EOGenericRecord when you don't need to define special behavior for your class.

- Create a custom class that uses the default implementation of key-value coding. If you plan to create a custom class, you must type its name in the Class field.

You can also use EOModeler to generate source code for your class; the resulting source files include definitions of instance variables and accessor methods that can be used by key-value coding. See "Generating Source Files" on page 81.

For more information on key-value coding and implementing enterprise object classes, see the chapter "Designing Enterprise Objects" in the book *Enterprise Objects Framework Developer's Guide*.

Generating Source Files

Once you finish specifying an enterprise object class, you can generate source files for it. However, at this stage of the development process, you may want to first create your project and design your application's user interface. Once you've created a project using Project Builder and included a model file in it, you can generate your source files and save them into the project.

You can create your enterprise object classes in either Objective-C or Java. To create an Objective-C class, use Property ▶ Generate Obj-C Files. To create a Java class, use Property ▶ Generate Java Files.

Additionally, there's a Property ▶ Generate Client Java Files. This command generates a Java class for use in the client-side of a Java Client web application.

Each command is described in more detail in the following sections.

Generating Objective-C Source Files

To generate Objective-C code files for your enterprise object class:

1. In the Model Editor, select the entity for which you have specified a class in the Entity Inspector.

   EOModeler only permits you to create source files for entities for which you have specified a custom enterprise object class. In other words, you can't generate source files for EOGenericRecord.

2. Choose Property ▶ Generate Obj-C Files or click the  button in the toolbar.

   EOModeler displays a Choose Class Name panel. If the model file is in a project, the Choose Class Name panel displays the project as the default destination.

3. Choose a destination, supply a name for the files if you want, and click Save.

   If you don't supply a name, the source files are named after the enterprise object class for which they are being generated and are given the appropriate extensions.

   If you opened the model file from a project, an additional panel appears, confirming that you want to insert the files in your project.

   Also, if you generate source files for an entity and files of the same name already exist, a panel is displayed asking if you want to cancel, overwrite, or merge the files. If you choose merge, the File Merge application starts with the old and new files displayed. You can then merge the files.

The end result of the Generate Obj-C Files command is:

- A header (**.h**) file that declares instance variables for all of the entity's class properties and declares accessor methods for those instance variables.
- An implementation (**.m**) file that provides basic implementations for the accessor methods.

For example, suppose you define an enterprise object class Movie. The instance variable declarations in the generated header file might resemble the following:

```
NSString *category;
NSCalendarDate *dateReleased;
NSDecimalNumber *language;
NSString *posterName;
NSDecimalNumber *revenue;
NSString *title;
id plotSummary;
Studio *studio;
NSMutableArray *directors;
NSMutableArray *roles;
```

Note that:

- The name of the generated class is the name you specified for the entity's Class Name characteristic.

- Instance variables that correspond to attributes are declared to be of the Objective-C value class specified in the model. For example, **revenue** is declared as an NSDecimalNumber and **dateReleased** is declared as an NSCalendarDate.

- Instance variables that represent to-one relationships are declared to be of type **id** for destination objects represented with EOGenericRecord (such as **plotSummary**) or as instances of the custom class used to represent them (such as **studio**).

- Instance variables that represent to-many relationships (such as **directors**) are NSMutableArrays.

The corresponding implementation (**.m**) file for Movie includes an implementation for each of the accessor methods declared in the header file. For example, the methods for setting and returning the value of the instance variable **title** are:

```
- (void)setTitle:(NSString *)value
{
  [self willChange];
  [title autorelease];
  title = [value retain];
}
- (NSString *)title { return title; }
```

Generating Java Source Files

Generate Java Files is similar to generating Objective-C files. To generate a Java (**.java**) file for your enterprise object class, follow the steps in "Generating Objective-C Source Files" on page 81, except that you choose Property ▶ Generate Java Files or click the  button in the toolbar.

As with the Objective-C source files, a Java source file declares instance variables and provides basic implementations of accessor methods for those variables. And in Java:

- The name of the generated class is the name you specified for the entity's Class Name characteristic.

- Instance variables that correspond to attributes are declared to be of the Java value class specified in the model, which can be String,

NSGregorianDate, Number, BigDecimal, NSData or a custom value class that you specify.

- Instance variables representing to-one relationships are declared to be of type EOEnterpriseObject for destination objects represented with EOGenericRecord or as instance of the custom class used to represent them. (EOEnterpriseObject is a Java interface defining basic enterprise object behavior; for more information, see the chapter "Designing Enterprise Objects" in the book *Enterprise Objects Framework Developer's Guide* or the EOEnterpriseObject interface specification in *Enterprise Objects Framework Reference*).

- As with Objective-C, instance variables that represent to-many relationships (such as **directors**) are NSMutableArrays.

Generate Client Java Files

Generate Client Java Files is similar to the other source generation commands. However, you only use this command when you're creating enterprise object classes to run on the client-side of a Java Client web application. (For a description of a Java Client application, see the chapter "What's Enterprise Objects Framework" in the book *Enterprise Objects Framework Developer's Guide*).

Using this and one of the other two source generation commands (either Objective-C or Java), you can create two versions of your enterprise object class. The different versions can have different class properties. For example, for security reasons, you might want to include social security number attribute in the server-side version of an enterprise object but exclude it from the client-side version.

To enable this distinction, you can specify whether an attribute is a Class Property () to be included in the server-side enterprise objects and also whether an attribute is a Client-Side Class Property () to be included in the client-side objects. For more information on setting these characteristics, see the chapter "Working with Attributes" on page 37.

To generate a Java (**.java**) file for a client-side version of your enterprise object class, follow the steps in "Generating Objective-C Source Files" on page 81, except that you choose Property ▶ Generate Client Java Files.

As with the source files generated by the other commands, a Java source file declares instance variables and provides basic implementations of accessor methods for those variables. The client-side Java files are generated exactly like the server-side Java files, except:

- The name of the generated class is the name you specified for the entity's Client-Side Class Name characteristic. If you haven't specified a Client-Side Class Name, it uses the name you specified for the entity's Class Name characteristic.
- If the model file is in a project, the default destination for the files is the ClientSideJava.subproj. Consequently, you can name your server-side and client-side enterprise object classes with the same name. The two versions of your class reside in different name spaces at runtime, too. Even if your server-side class is a Java class, it descends from com.apple.*yellow*.eocontrol.EOCustomObject, while the client-side class descends from com.apple.client.*eocontrol*.EOCustomObject.
- Instance variables are generated only for attributes and relationships that are set as Client-Side Class Properties.

Customizing Source File Generation

When you create a project with the type "EOApplication," it inserts three files into the project's Supporting Files suitcase: **EOInterfaceFile.template**, **EOImplementationFile.template**, and **EOJavaClass.template**. You can use these files to customize your **.h**, **.m**, and **.java** file output, respectively. In their unmodified form these files match the source file generation scheme used by EOModeler.

Creating a Subclass

Enterprise Objects Framework supports mapping database tables to inheritance hierarchies of enterprise object classes using three different approaches. For one of the approaches—single table mapping—EOModeler provides support to help you model the mapping. In the single table mapping, all the enterprise object classes in the inheritance hierarchy map to the same database table; each class makes use, however,

of different sets of the tables columns. Consequently, you need to create an entity for each enterprise object class in the hierarchy, and each of the entities map to the same table.EOModeler facilitates this by creating "subclass entities" and setting up the parent-child relationships for you.

To create a "subclass" entity, select the entity you want to use as the parent and choose Property ▶ Create Subclass. A new entity is created that maps to the same database table as the parent entity.

For more discussion of inheritance, see the chapter "Advanced Enterprise Object Modeling" in the book *Enterprise Objects Framework Developer's Guide*.

Setting Other Information for an Entity

From the Entity Inspector you can navigate to other Inspectors to specify additional information for your entity.

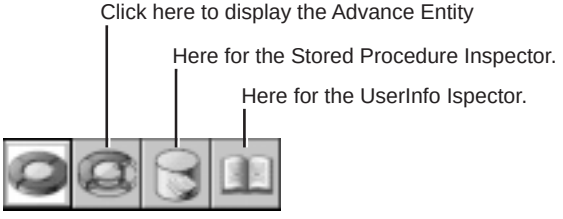


Figure 36. Icons for Navigating to Other Inspectors

Advanced Entity Inspector

The Advanced Entity Inspector lets you set more complex behavior for your entity, such as that to support inheritance.

To display the Advanced Entity Inspector, select the Advanced Entity Inspector icon at the top of the Entity Inspector.

Figure 37. Advanced Entity Inspector

Batch Faulting Size

The Batch Faulting Size field lets you set the number of EOFaults that should be triggered when you first access an object of this type. By default, only one object is fetched from the database when you trigger an EOFault. By providing a number *N* in this field, you specify that *N* other EOFaults of the same entity should be fetched from the database along with the first one. This improves performance by minimizing round trips to the database server. For more information, see the chapter "Answers to Common Design Questions" in the book *Enterprise Objects Framework Developer's Guide*.

External Query

The External Query field allows you to specify any SQL statement that will be executed as is (that is, you can't perform any substitutions) when EOF does an unqualified fetch. On Sybase this can be a stored procedure. The columns selected by this SQL statement must be in alphabetical order by internal name, and must match in number and type with the class properties specified for the entity.

Qualifier

This field is used to specify a restricting qualifier. A restricting qualifier maps an entity to a subset of rows in a table. Restricting qualifiers are commonly used when you're using single table inheritance mapping, in which the data for a class and its subclasses is all stored in a single table. When you add a restricting qualifier to an entity, it causes a fetch for that entity to only retrieve objects of the appropriate type. For example, the Rentals sample database has a MOVIE_MEDIA table that includes rows for both the VideoTape and LaserDisk entities. VideoTape has the restricting qualifier (media = 'T'), and LaserDisk has the restricting qualifier (media = 'D'). When you fetch objects for the entity VideoTape, only rows that have the value 'T' for the attribute **media** are fetched. For more discussion of single table and other types of inheritance mapping, see the chapter "Advanced Enterprise Object Modeling" in the book *Enterprise Objects Framework Developer's Guide*.

Parent

You use this field to specify a parent entity for the current entity. This field is used to model inheritance relationships. For example, in the Rentals database, the Customer entity is the parent of the Member and Guest entities (since Members and Guests are types of Customers). For more information, see the chapter "Advanced Enterprise Object Modeling" in the book *Enterprise Objects Framework Developer's Guide*.

Read Only

The Read Only checkbox indicates whether the data that's represented by the entity can be altered by your application.

Cache In Memory

The Cache In Memory checkbox lets you specify that the objects associated with an entity should be cached in memory for quick access. Caching an entity's objects allows Enterprise Objects Framework to evaluate queries in memory, thereby avoiding round trips to the database. This is most useful for read-only entities, where there is no danger of the cached data getting out of sync with database data. This technique should only be used with small tables, since it fetches the entire table into memory.

Abstract

The Abstract checkbox indicates whether the entity is abstract. An abstract entity is one for which no objects are ever instantiated in your application. For example, in the Rentals database, the Customer entity is abstract since Customer objects are never instantiated (though objects of its sub-entities, Member and Guest, are). Like the Parent field, this option is used to model inheritance. For more information, see the chapter "Advanced Enterprise Object Modeling" in the book *Enterprise Objects Framework Developer's Guide*.

Stored Procedures Inspector

You use the Stored Procedures Inspector to specify stored procedures that should be executed when a particular database operation (such as insert or delete) occurs. You type the name of the stored procedure in the field associated with the database operation. Stored procedures are read from the database when you create a new model and included in its **.eomodeld** file. You can also add stored procedures through EOModeler after the fact.

Creating stored procedures and assigning them to entities is described in detail in the chapter "Working with Stored Procedures" on page 91.

UserInfo Inspector

You use the UserInfo Inspector to add key-value pairs to the UserInfo NSDictionary. The UserInfo dictionary provides a mechanism for extending your model. You can use it to define custom behavior for an entity. For example, you can put information in the UserInfo dictionary to be used by delegate methods that perform operations on the entity.

Chapter 7

Working with Stored Procedures

This chapter describes working with stored procedures in your model. It covers adding stored procedures as well as assigning them to entities for common operations in the sections:

Adding Stored Procedures

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:

1. Select the Stored Procedures icon in the tree view.
2. Choose Property ▶ Add Stored Procedure.
3. Specify a name and external name for the stored procedure.

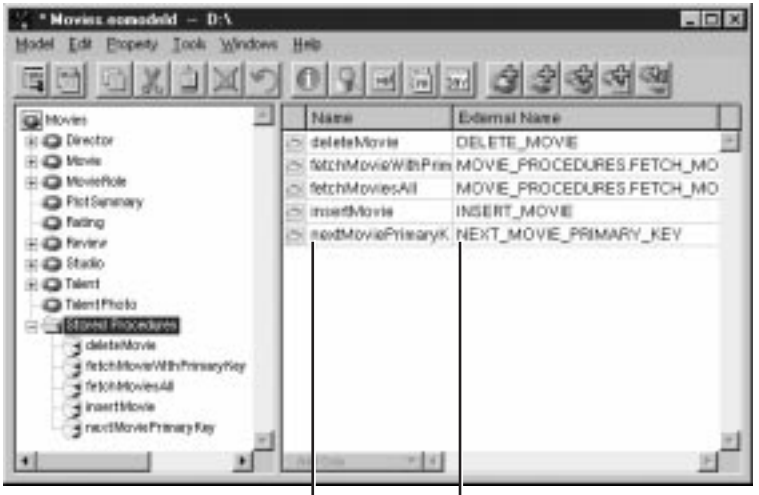


Figure 38. Adding a Stored Procedure

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 a stored procedure argument.

To define and display the attributes of a stored procedure:

1. Select the stored procedure in the tree view.

   Alternatively, you can double-click the  icon to the left of a stored procedure in the Model Editor's stored procedure table.

2. Choose Property ▶ Add Argument.

3. Specify the argument's characteristics in the Model Editor's table.

Minimally, you must provide values for the Name, Column, Direction, External Type, and Value Class characteristics.

Each table column corresponds to a single characteristic of a stored procedure argument. By default, the columns included in the table only represent a subset of the possible characteristics you can set for a given entity. To add columns for additional characteristics, use the Add Column menu in the lower left corner of the table.

The following table describes the characteristics you can set for a stored procedure argument.

Characteristic	What it is
Allows Null	Indicates whether the argument's value can be NULL.
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 (Java)	The Java type to which the argument value will be coerced in your application.
Value Class (Obj-C)	The Objective-C type to which the argument value will be coerced in your application.
Value Type	The format type for NSNumber classes such as "i" or "d".
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 following section discusses this requirement more thoroughly.

Assigning a Stored Procedure to an Entity

You can assign stored procedures to entities to be used to perform the following operations:

- Insert a new object.
- Delete an object.
- Fetch all the objects for an entity.
- Fetch an object by its primary key.
- Generate a primary key value for a new object.

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 in the previous section.
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 (see the chapter "Answers to Common Design Questions" in the book *Enterprise Objects Framework Developer's Guide*).

To assign a stored procedure to an entity in EOModeler:

1. Select the entity with which you want to associate a stored procedure.
2. Open the 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 39. 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. For more information on these requirements, see the chapter "Answers to Common Design Questions" in the book *Enterprise Objects Framework Developer's Guide.*

Chapter 8

Working with Fetch Specifications

You can use EOModeler to create a query—called a *fetch specification*—name it, and store it in the model file. You can then use these pre-defined queries in your application—invoking them programmatically or binding them to your application's user interface. This chapter describes creating fetch specifications in EOModeler. It's organized in the following sections:

Fetch Specifications

To perform a query in an Enterprise Objects Framework application, you use an EOFetchSpecification object. Fetch specifications have associated with them an entity, a qualifier, a sort ordering, and several other options. You can create fetch specifications programmatically, or you can use EOModeler to create and store them. For more information on the EOFetchSpecification class, see its specification in the *Enterprise Objects Framework Reference*.

Creating a Fetch Specification

To create a fetch specification in EOModeler:

1. Select the entity with which the fetch specification will be associated.

2. Choose Property ▶ Add Fetch Specification or click the  button in the tool bar.

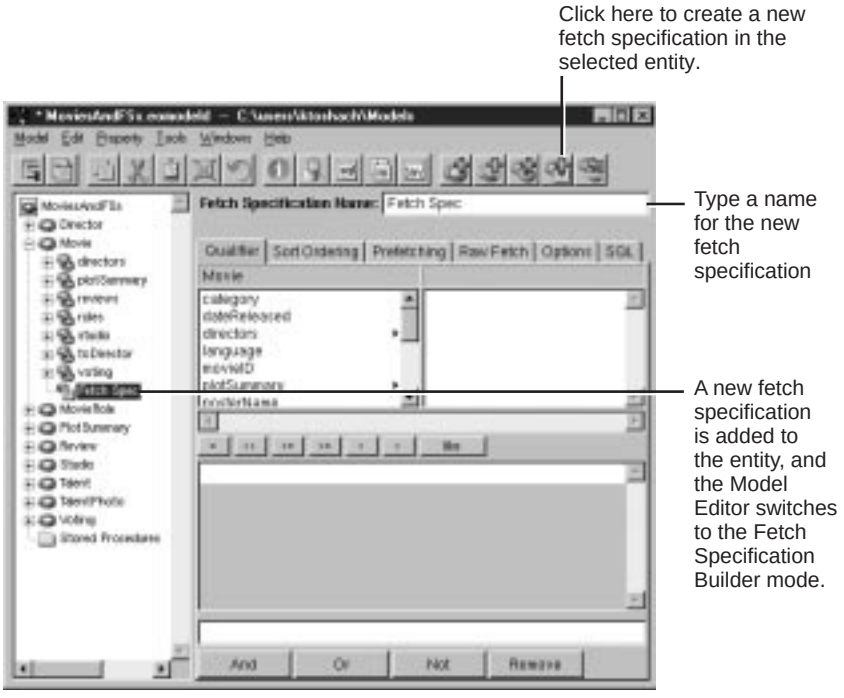


Figure 40. Adding a Fetch Specification to an Entity

3. Type a name for the fetch specification in the Fetch Specification Name field.

There are many different ways to configure the fetch specification. The most common way is to build a qualifier for the fetch specification to fetch with. For more information, see the section "Building a Qualifier" on page 103. Alternatively, you can also configure a fetch specification to fetch using custom SQL or a stored procedure. For more information, see "Using Custom SQL and Stored Procedures" on page 115.

In addition to specifying how a fetch specification retrieves its data, you can specify other options, such as sort orderings and performance tuning settings. The following sections describe the possible configurations and their uses.

Building a Qualifier

EOModeler's Fetch Specification Builder mode provides an interface for building a qualifier graphically. To use it, select your fetch specification, and choose the Qualifier tab in the Fetch Specification Builder.

For example, a Movie has a Voting object that keeps a **runningAverage** of how reviewers have voted on the movie. Suppose you want to create a fetch specification for the Movie entity that fetches all movies whose **runningAverage** is greater than eight. To build the qualifier for such a query:

1. In the attribute browser, click Movie's **voting** relationship.

2. In the second column of the browser, select the **runningAverage** attribute.

   The text field just under the browser is updated to display the attribute as the left hand side of an expression. Note that the **runningAverage** attribute is represented by a *key path* (**voting.runningAverage**) that identifies the relationship (**voting**) through which the attribute is accessed.

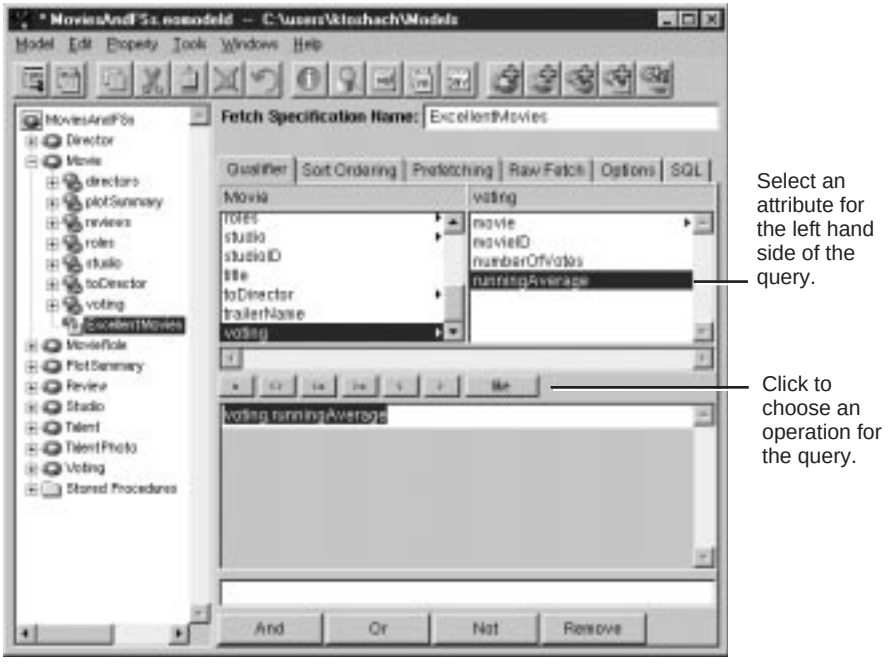


Figure 41. Building an Expression.

3. Click the >= button.

   EOModeler adds a greater than or equal to operator to the expression.

4. In the text field, type “8” as the right hand side of the expression.

Instead of building the expression by choosing attributes in the attribute browser and by clicking an operator button, you can type directly into the text field for the expression.

Creating Compound Qualifiers

You can also use the Query Builder to create compound qualifiers made up of multiple expressions. Click the And button to create a new expression and AND it with the first one. Click the Or button to create a new expression and OR it with the first.

For example, building on the example in the previous section, suppose that you want to fetch movies with a **runningAverage** greater than or equal to eight but that also have at least ten voters contributing to the average. To further restrict the fetch specification:

1. Click the And button.

   EOModeler adds a second expression and ANDs it to the first expression.

2. Choose **voting.numberOfVotes** in the attribute browser.

3. Click the >= button.

4. Type 10 as the right hand side of the expression.

   As you build up a complex query, the text field at the bottom of the Query Builder updates to include the full text of the compound qualifier. Instead of building up expressions one by one with the And and Or buttons, you can type directly into this lower text field. The Qualifier Builder parses the qualifier string and displays the individual expressions.

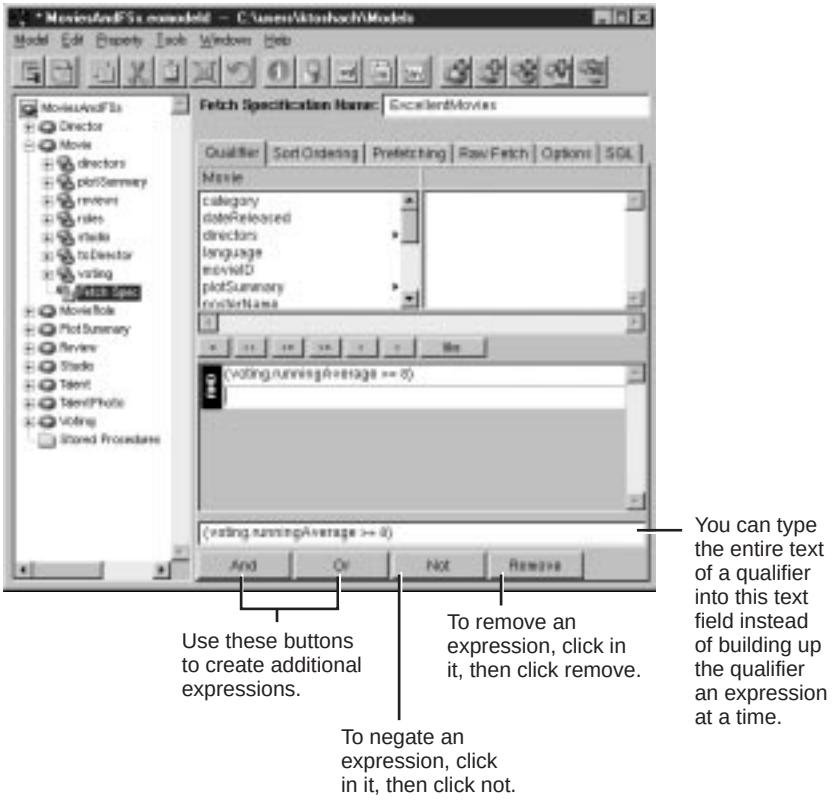


Figure 42. Creating a Compound Qualifier

To negate an expression, click in the text of the expression you want to negate, and then click the Not button. Similarly, to remove an expression click in the text of the expression you want to negate, and then click the Remove button.

Using Qualifier Variables

You can specify absolute criteria for a fetch specification's qualifier—"voting.runningAverage >= 8", for example. However, such a fetch specification is of limited use. More frequently, you want to specify the form of a qualifier and let users supply specific values when they run the application. You can do this with *qualifier variables*.

You specify a qualifier variable using the dollar sign character ($), as in the following:

```
dateReleased = $aDate
```

For example, suppose you want to allow users to search for movies by title, date released, or studio. The query would look like this:

```
((title = $title) OR
(dateReleased = $date) OR
(studio.name = $studioName))
```

You can build this qualifier in EOModeler as specified in the previous sections, and then bind its qualifier variables (**title**, **date**, **studioName**) to your application's user interface. When the application runs, Enterprise Objects Framework automatically replaces the qualifier variables with values supplied in the user interface. You can set this up as follows:

1. Create the fetch specification with EOModeler.

2. Use Query Builder's user interface to set up a query on the **title**, **date**, and **studio.name** attributes.

   On the right side of each expression, use the $ syntax to identify the qualifier variables.

Depending on the type of graphical user interface your application uses, you access the fetch specification's query bindings differently. For example, in WebObjects Builder, you access the query bindings in the following way:

3. Use WebObjects builder to create a component that allows the user to enter the query criteria.

   You might create text fields for the title and date released and a pop-up list for the studio name, for example.

4. Drag the fetch specification from EOModeler into your component.

   This has the effect of creating a new display group for your specification's entity.

5. In the WebObjects Builder panel that opens, choose "Add and Configure."

6. Configure the new display group, setting its fetch specification to the one you defined in your model.

7. In WebObjects Builder, bind the user interface elements to the **queryBindings.title**, **queryBindings.date**, and **queryBindings.studioName** keys of your display group (**movieDisplayGroup**, for example).

In Interface Builder, the steps are similar except that you bind the user interface elements to **@bindings.title**, **@bindings.date**, and **@bindings.studioName** keys of your display group. The @bindings syntax represents the value associated with the named qualifier variable.

Assigning a Sort Ordering

To specify the order in which the fetch specification fetches its objects, use the Sort Ordering tab in the Fetch Specification Builder, as shown in Figure 43.

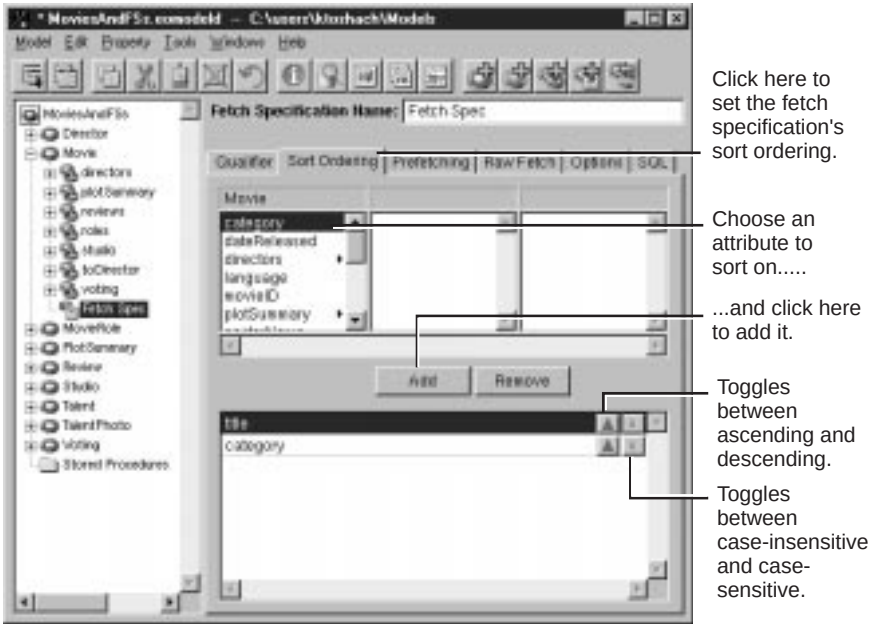


Figure 43. Assigning a Sort Ordering

Simply choose an attribute to sort on, and click Add. The order in which you add the attributes specifies the weight to assign to them. In Figure 43, the fetch specification sorts first on title and then on category.

Additionally, for each attribute you sort on, you can specify an ascending or descending order and whether to perform a case-sensitive or case-insensitive comparison.

Specifying Prefetching and Other Options

There are numerous options you can configure to tune a fetch specification's behavior. This section describes the options on the Prefetching and Options tabs.

Configuring Prefetching

Use this tab to identify relationships that should be fetched along with the objects specified by the fetch specification. For example, when fetching Rental objects, you can prefetch associated Fees and Units. Doing so forces a rental's fees and unit to be retrieved along with the rental itself (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.

To specify a relationship to prefetch, select the relationship in the Fetch Specification Builder's browser and click Add, as shown in Figure 44.

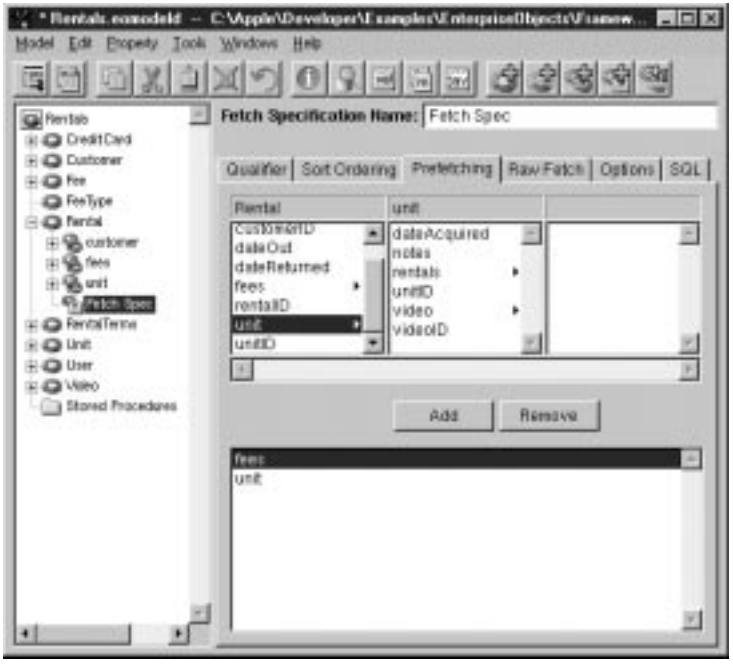


Figure 44. Specifying Relationships to Prefetch

Setting a Fetch Limit

To specify the maximum number of objects to fetch with a fetch specification, go to the Options tab of the Fetch Specification Builder, as shown in Figure 45. There you can specify the maximum number of objects to fetch. The default fetch limit is zero, indicating that there is no fetch limit. Type a number in the Max Rows text field to specify a maximum number.

Use the "Prompt on limit" box to specify what the Framework should do when the fetch limit is reached. If the box is checked, the Framework prompt the user about whether to continue fetching after the maximum has been reached. If the box isn't checked, the Framework simply stops fetching when it reaches the limit.

For more information on managing fetch limits, see the EOFetchSpecification class description and the EOEditingContext EOMessageHandlers interface description in the *Enterprise Objects Framework Reference*.

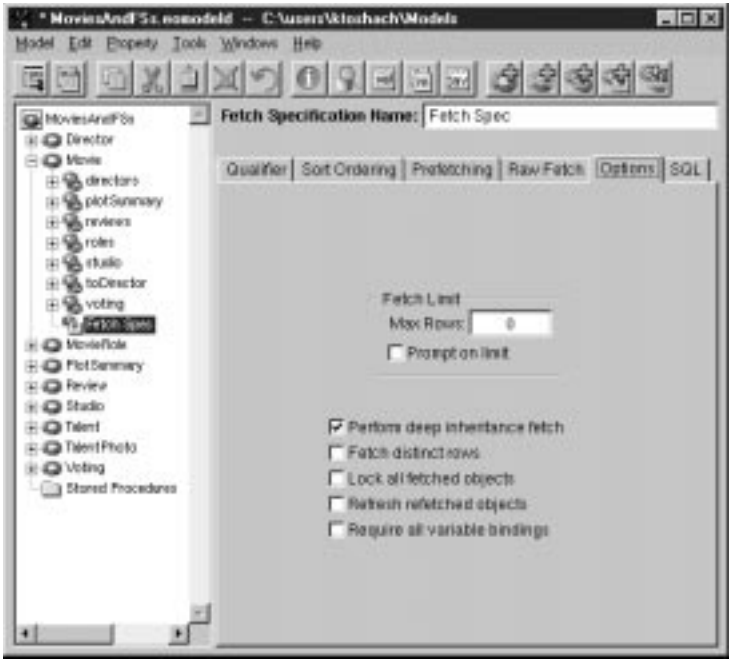


Figure 45. The Options Tab

Other Options

The other options on the Options tabs are explained below:

Perform deep inheritance fetch

An indicator of whether to fetch deeply or not. This is used with inheritance hierarchies when fetching for an entity with sub-entities. A deep fetch produces all instances of the entity and its sub-entities, while a shallow fetch produces instances only of the entity in the fetch specification.

Fetch distinct rows

An indicator of whether to produce distinct results or not. Normally if a record or object is selected several times, such as when forming a join, it appears several times in the fetched results. A fetch specification that fetches distinct rows filters out duplicates so that each record or object selected appears exactly once in the result set.

Lock all fetched objects

If a fetch specification locks fetched objects, it locks each object as it selects it.

Refresh refetched objects

If a fetch specification refreshes refetched objects, existing objects are overwritten with newly fetched values when they've been updated or changed. With fetch specifications that don't refresh, existing objects aren't updated when their data is refetched (the fetched data is simply discarded).

Require all variable bindings

Specifies whether a missing value for a qualifier variable is ignored or whether the Framework requires that each qualifier variable have a value assigned to it. If "Require all variable bindings" is checked, the Framework throws an exception during variable substitution. If it isn't checked, any qualifier nodes for which there are no variable bindings are pruned from the qualifier.

Configuring Raw Row Fetching

When you perform a fetch in an Enterprise Objects Framework application, the information from the database is fetched and stored in a graph of enterprise objects. This object graph provides many advantages, but it can be large and complex. If you're creating a simple application, you may not need all of the benefits of the object graph. For example, a WebObjects application that merely displays information from a database without ever performing database updates and without ever traversing relationships might be just as well served by fetching the information into a set of dictionaries rather than a set of enterprise objects.

Enterprise Objects Framework 3.0 supports this concept of a simplified fetch, called *raw row* fetching. In raw row fetching, each row from the database is fetched into an NSDictionary object.

When you use raw row fetching, you lose some important features:

- The NSDictionary objects are not uniqued.
- The NSDictionary objects aren't tracked by an editing context.
- You can't access to-many relationship information. (To access to-one relationship information, you use key paths such as "movie.dateReleased".)

To set up raw row fetching, go to the Raw Fetch tab of the Fetch Specification Builder as shown in Figure 46.

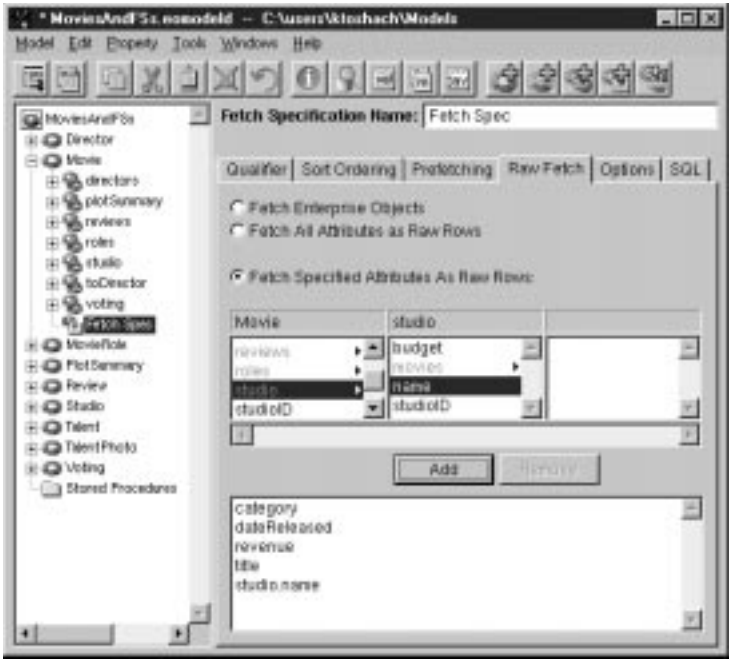


Figure 46. Specifying a Raw Row Fetch

Using Custom SQL and Stored Procedures

Instead of building a qualifier for the fetch specification to fetch with, you can specify custom SQL or a stored procedure. To do so, use the SQL tab of the Fetch Specification Builder, as shown in Figure 47.

Figure 47. Using a Custom SQL Expression or a Stored Procedure

To use custom SQL, check the "Use Raw SQL Expression" box, and provide the SQL in the text field just below the box. If you've built a qualifier in the Qualifier Builder, this text field is initialized with the corresponding SQL. Checking the "Use Raw SQL Expression" box enables this text field so you can modify the text. Note that the Fetch Specification Builder isn't able to parse arbitrary SQL to produce a corresponding qualifier in the Qualifier Builder.

To use a stored procedure, check the "Use Stored Procedures" box, and choose the stored procedure from the list just below the box. The stored procedure must be defined in the model.

Testing a Fetch Specification

You can test a fetch specification using EOModeler's data browser. Click the fetch specification in the Model Editor's tree view, and then open the Data Browser. EOModeler connects to the database, invokes the fetch specification, and displays the results.

If the fetch specification includes qualifier variables, EOModeler displays a Qualifier Bindings panel with which you can supply values to bind to the variables ().

Chapter 9

Interacting with a Database

This chapter describes the ways you can use EOModeler to interact with your database.

Setting Adaptor Information

A model includes a connection dictionary, which contains the information needed to connect to a database server. The keys of the connection dictionary identify the information the server expects, and the values associated with those keys are the values that the adaptor tries when logging into the database.

When you initialize an adaptor from a model, any connection information stored with the model is copied into the adaptor object.

The connection dictionary contains the last values you entered in the login panel and saved as a part of your model (so long as you haven't manually edited the connection dictionary in your model file). You can change the connection dictionary's values from EOModeler; this is called setting adaptor information.

To set adaptor information:

1. Choose Model ▶ Set Adaptor Info.

   EOModeler displays a login panel that contains values taken from the model's connection dictionary.

2. In the login panel, make the edits you want reflected in your connection dictionary, and click OK.

For example, if you specified a user name and password to log into a database and create your model, you can remove that information from the connection dictionary by clearing those fields in the login panel. Then, in your application, you can prompt the user for a user name and password by sending a **runLoginPanelAndValidateConnectionDictionary** message to your adaptor object.

You can also edit the connection dictionary in its raw form using the Connection Dictionary Inspector. This provides you access to

connection dictionary entries that the login panel doesn't configure. To use the Connection Dictionary Inspector, select the model icon in the Model Editor, and display the Inspector.

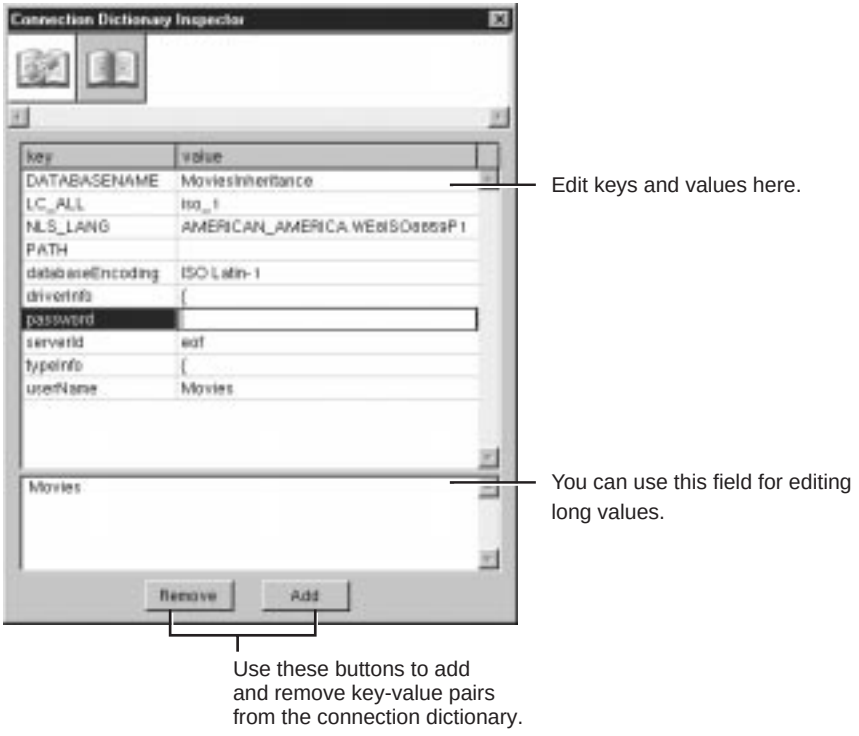


Figure 48. Connection Dictionary Inspector

For more discussion of how Enterprise Objects Framework manages database connections and connection dictionaries, see the chapter "Connecting to the Database" in the book *Enterprise Objects Framework Developer's Guide*.

Switching Adaptors

You can change the database and adaptor your model is based on. To do so:

1. Choose Model ► Switch Adaptor.

   This displays a New Model panel listing all the available adaptors.

2. Select the adaptor you want to switch to and click OK.

   EOModeler displays the login panel for the database that corresponds to the adaptor you selected.

3. Fill in the login panel and click OK.

When you switch adaptors, Enterprise Objects Framework automatically updates the mapping between the internal and external (database) types to work with the new adaptor's database.

Using the Data Browser

You can use the Data Browser to display database records associated with an entity in the Model Editor.

To display an entity's records in the Data Browser, select an entity or a fetch specification and choose Tools ► Data Browser.

To browse the records associated with a different entity, select the entity icon in the Model Editor and drag it into the Entity well of the Data Browser.To view a subset of the attributes for an entity, select one or more attributes and choose Tools ► Data Browser.

You can rearrange the columns in the Browser by dragging their column heads to new positions. You can also resize columns by selecting their column heads and dragging the edges until the column is the desired size.

You can change the sorting order of the Browser by using the buttons in the lower right corner. By default, the data is displayed according to how it was returned from the database. You can sort on the first column in either ascending or descending order by clicking the appropriate sort button. So, for example, to sort the records alphabetically by the movie name in the Movie database, drag the **title** column into the first column of the Browser and click the ascending sort button. To restore the order of the data as it was returned from the database, click the default order button.

Generating SQL

You can use EOModeler to create a model from scratch (that is, to create a model that's not initialized from an existing database), and then use that model to generate the SQL necessary to create a database. You can also edit a model for an existing database and generate SQL statements from the model that can be used to regenerate the database with the new settings.

To generate SQL for one or more entities, select the entities and choose Property ► Generate SQL.