Chapter 3

Using EOModeler

You use the EOModeler application to create models. A model defines, in entity-relationship terms, the mapping between a relational database and enterprise object classes.

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 template source code files for the enterprise object classes you specify.
- Generate SQL that can then be used to create database tables.

A model enables an enterprise object to remain synchronized with a corresponding database row throughout the execution of your application. Models are fully accessible to your application; at run time you can dynamically generate new models or change the mapping in existing models.

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 for improved performance. A model actually 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 a window listing the individual entity files in the model. Each of these files has a .plist extension, indicating that the files' contents are in ASCII property list format. You can view the file for a particular entity 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
Column	EOAttribute	Enterprise object class instance variable (class property)
Referential Constraint	EORelationship	Pointer to another object
Row	—	Enterprise object instance

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 file 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 "Adding Derived and Flattened Attributes" on page 113.

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.

Launching EOModeler

The EOModeler application is located in the OPENSTEP Enterprise Software program group on Windows NT (and in the /NextDeveloper/Apps directory on Mach) and is represented by the icon shown in Figure 6.

Figure 6. *EOModeler's Application Icon*

Launch the application by double-clicking the icon, or by double-clicking an existing model file.

Creating a New Model

To create a model:

In EOModeler, choose Model m New.

EOModeler displays a panel prompting you to select an adaptor, as shown in Figure 7.

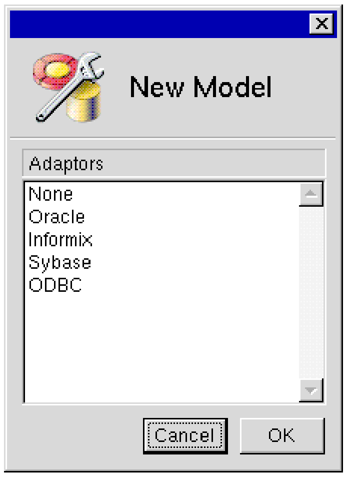
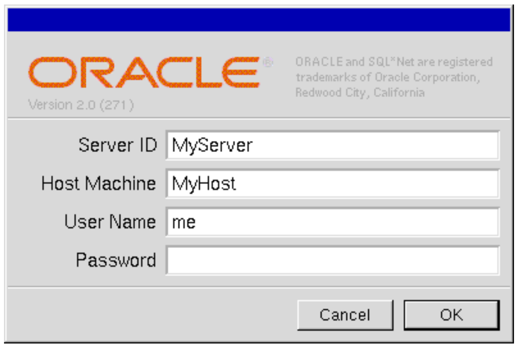


Figure 7. *Selecting an Adaptor*

Select the adaptor you want to use and click OK.

EOModeler displays the login panel for the database that corresponds to the selected adaptor. The examples in this chapter use the Oracle version of the Movie database included with the Enterprise Objects Framework; Figure 8 shows the Oracle login panel.


ORACLE
Version 2.0 (271)
ORACLE and SQL*Net are registered trademarks of Oracle Corporation, Redwood City, California
Cancel
OK

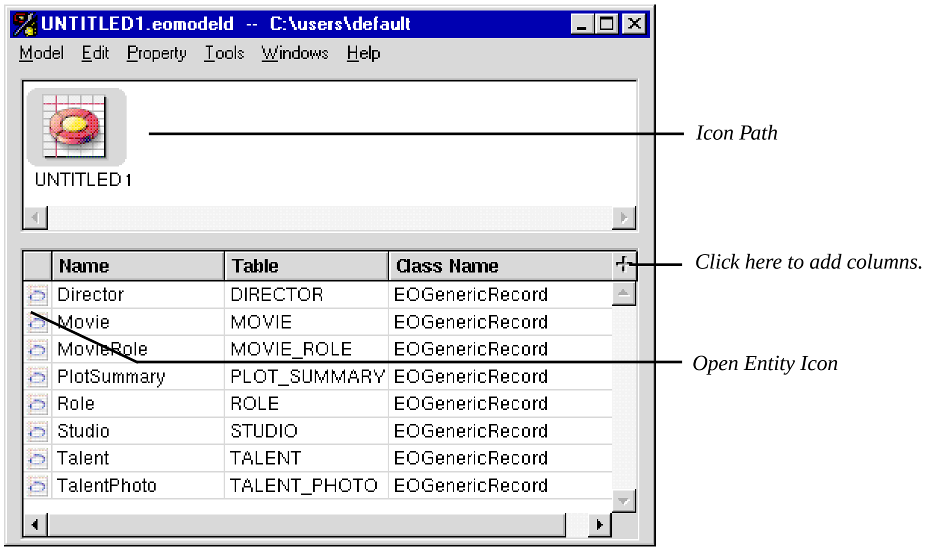
Server ID	MyServer
Host Machine	MyHost
User Name	me
Password	

Figure 8. *Oracle Login Panel*

Fill in the login panel and click OK.

Using the Model Editor in Table Mode

When you first log in to a database, EOModeler uses an adaptor to read the data dictionary from the database and create the original model. This model is displayed in the Model Editor, shown in Figure 9, which lists the entities available for the database you specified in the login panel. EOModeler uses the table mode of the Model Editor to display the new model. You can also use browser mode in the Model Editor—for more information see "Using the Model Editor in Browser Mode" on page 100.



Name	Table	Class Name
Director	DIRECTOR	EOGenericRecord
Movie	MOVIE	EOGenericRecord
MovieRole	MOVIE_ROLE	EOGenericRecord
PlotSummary	PLOT_SUMMARY	EOGenericRecord
Role	ROLE	EOGenericRecord
Studio	STUDIO	EOGenericRecord
Talent	TALENT	EOGenericRecord
TalentPhoto	TALENT_PHOTO	EOGenericRecord

Figure 9. *The Model Editor in Table Mode*

Icon Path

The icon path changes to indicate your current location as you navigate around a model. For example, in Figure 10, the icon path indicates that the current selection is the lastName attribute in the Talent entity, which is part of the Movies model.

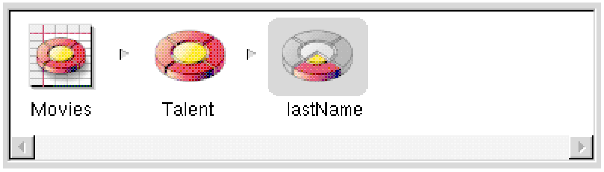


Figure 10. *Icon Path*

You can click on any icon in an icon path to navigate to that part of the model. You also use the icons in the icon path in drag and drop operations—for example, to drag an entity into the Data Browser (described below) or Interface Builder (described in the next chapter).

Open Entity Icon

When you double-click the icon to the left of an entity, it displays that entity's attributes.

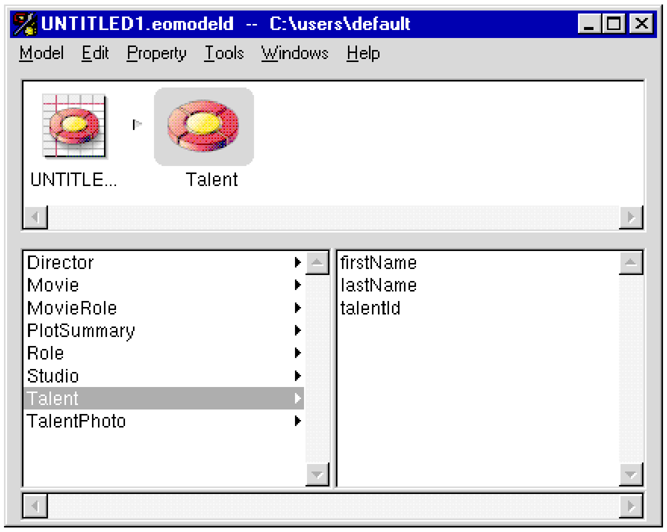
Menu

You use the menu to add columns for an entity. Each column represents a different characteristic you can set for an entity. By default, when you first run EOModeler the table mode has just four columns: Open Entity, Name, Table, and Class Name. The menu provides these additional items: Open Entity, Parent, and External Query. The following table describes the characteristics you can set for an entity.

Characteristic	What it is
Open Entity	Adds a column with the Open Entity icon, which you can double-click on to display an entity's attributes.
Name	The name your application uses for the entity. By default, EOModeler supplies names based on the name of the corresponding table in the database.
Table	The name of the database table that corresponds to the entity.
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.
Parent	Indicates an entity's parent—used to model inheritance.
External Query	Any SQL statement that will be executed as is—on Sybase, this can be a stored procedure.

Using the Model Editor in Browser Mode

To see the Model Editor in browser mode, choose Tools m Browser Mode.



Director
Movie
MovieRole
PlotSummary
Role
Studio
Talent
TalentPhoto

firstName
lastName
talentId

Figure 11. *Model Editor in Browser Mode*

To display the attributes for a particular entity, such as Talent, select the entity. The attributes appear in the column to the right of the entity.

What a Default 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 the corresponding Objective-C type

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:

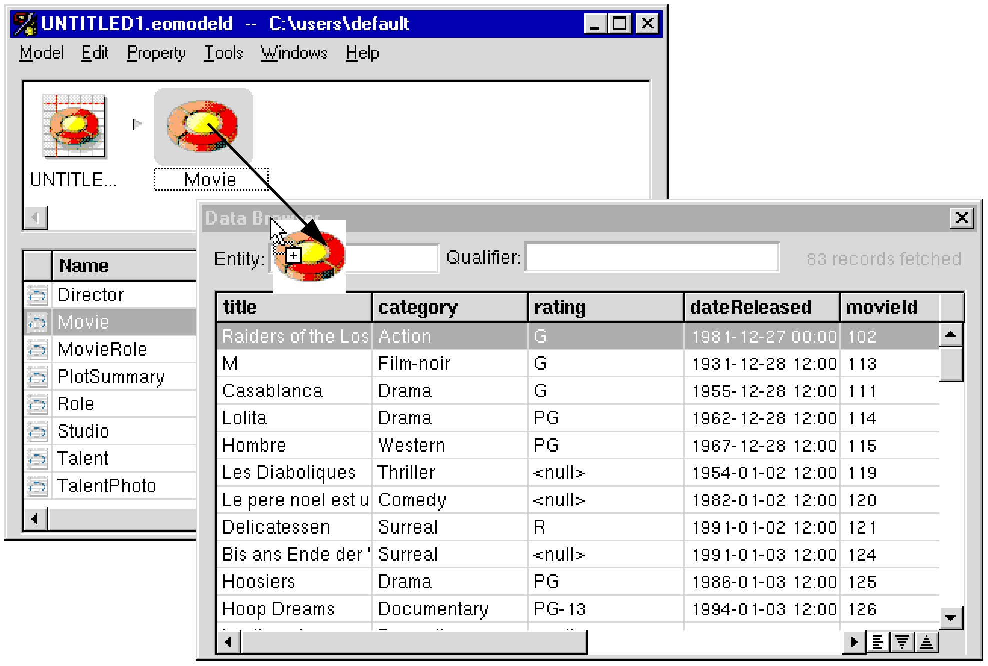
Database Column	Attribute Name
NAME	name
FIRST_NAME	firstName
MOVIE_ID	movieId

Using the Data Browser

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

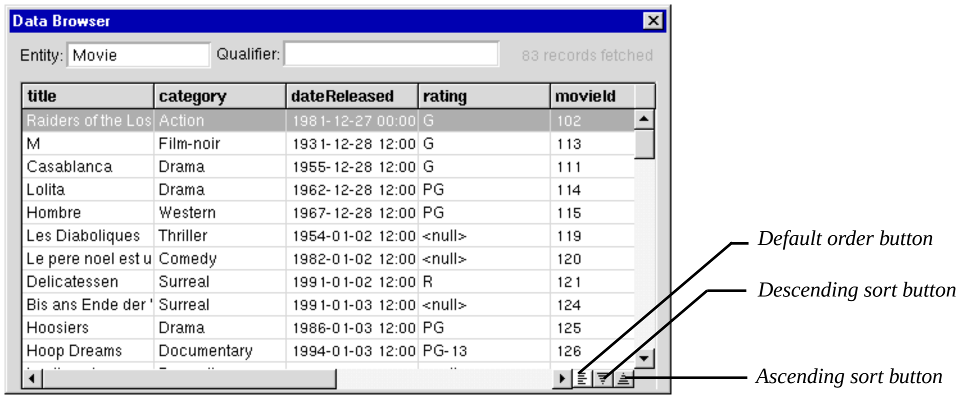
To display an entity's records in the Data Browser, select the entity and choose Tools m Data Browser.

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, as shown in Figure 12. To view a subset of the attributes for an entity, select one or more attributes and drag the associated icon into the Entity well.



title	category	rating	dateReleased	movieId
Raiders of the Los	Action	G	1981-12-27 00:00	102
M	Film-noir	G	1931-12-28 12:00	113
Casablanca	Drama	G	1955-12-28 12:00	111
Lolita	Drama	PG	1962-12-28 12:00	114
Hombre	Western	PG	1967-12-28 12:00	115
Les Diaboliques	Thriller	<null>	1954-01-02 12:00	119
Le pere noel est u	Comedy	<null>	1982-01-02 12:00	120
Delicatessen	Surreal	R	1991-01-02 12:00	121
Bis ans Ende der '	Surreal	<null>	1991-01-03 12:00	124
Hoosiers	Drama	PG	1986-01-03 12:00	125
Hoop Dreams	Documentary	PG-13	1994-01-03 12:00	126

Figure 12. *Dragging an Entity into the Data Browser*



title	category	dateReleased	rating	movieId
Raiders of the Los	Action	1981-12-27 00:00	G	102
M	Film-noir	1931-12-28 12:00	G	113
Casablanca	Drama	1955-12-28 12:00	G	111
Lolita	Drama	1962-12-28 12:00	PG	114
Hombre	Western	1967-12-28 12:00	PG	115
Les Diaboliques	Thriller	1954-01-02 12:00	<null>	119
Le pere noel est u	Comedy	1982-01-02 12:00	<null>	120
Delicatessen	Surreal	1991-01-02 12:00	R	121
Bis ans Ende der '	Surreal	1991-01-03 12:00	<null>	124
Hoosiers	Drama	1986-01-03 12:00	PG	125
Hoop Dreams	Documentary	1994-01-03 12:00	PG-13	126

Figure 13. *The Data Browser*

You can rearrange the columns in the Browser by dragging their title tabs to new positions. You can also resize columns by selecting their title tabs and dragging the tab 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. However, 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.

Inspecting and Modifying Attributes

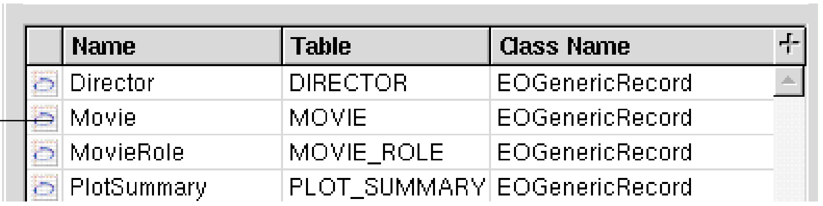
EOModeler provides two mechanisms for viewing and modifying your attributes: the table mode of the Model Editor, and the Attribute Inspector.

You can use either mechanism to examine the characteristics of the attributes in your model and make any necessary refinements. You can use the Model Editor for most common operations, but for some more sophisticated changes you need to use the Inspector.

Working with Attributes in Table Mode

To display an entity's attributes in table mode, double-click the icon to the left of an entity, as shown in Figure 14.

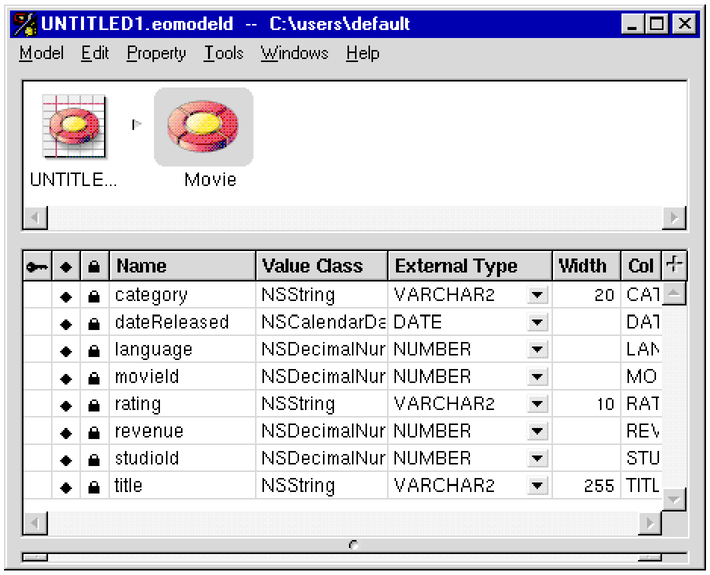
Double-click here



	Name	Table	Class Name
	Director	DIRECTOR	EOGenericRecord
	Movie	MOVIE	EOGenericRecord
	MovieRole	MOVIE_ROLE	EOGenericRecord
	PlotSummary	PLOT_SUMMARY	EOGenericRecord

Figure 14. *Double-click an Open Entity Icon to Display Attributes*

The display changes to show all of the entity's attributes, as shown in Figure 15.


UNTITLED1.eomodeld -- C:\users\default
Model Edit Property Tools Windows Help
UNTITLE...
Movie

			Name	Value Class	External Type	Width	Col
	◆	🔒	category	NSString	VARCHAR2	20	CAT
	◆	🔒	dateReleased	NSCalendarDa	DATE		DAT
	◆	🔒	language	NSDecimalNur	NUMBER		LAN
	◆	🔒	movieId	NSDecimalNur	NUMBER		MO
	◆	🔒	rating	NSString	VARCHAR2	10	RAT
	◆	🔒	revenue	NSDecimalNur	NUMBER		REV
	◆	🔒	studioId	NSDecimalNur	NUMBER		STU
	◆	🔒	title	NSString	VARCHAR2	255	TITL

Figure 15. *Displaying an Entity's Attributes*

You can use this view to modify an attribute's characteristics (alternatively, you can use the Attribute Inspector). Each 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 menu at the upper left corner of the table. The following table describes all of the characteristics for which you can add columns.

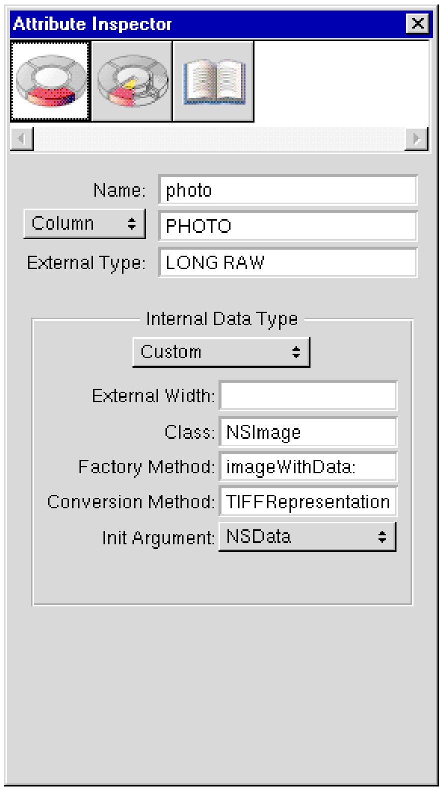
Characteristic	**What it is**	**How you modify it**
Primary Key	Declares whether a property is, or is part of, the primary key for the entity.	Click in the column to toggle primary key off and on.
Class Property	Indicates a property that meets both of these criteria: you want to include it in your class definition, and it can be fetched from the database.	Click in the column to toggle class property off and on.
Locking	Indicates whether an attribute should be used for locking when an update is performed.	Click in the column to toggle locking off and on.
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.
Value Class	The Objective-C type to which the attribute will be coerced in your application. EOModeler supplies a default mapping between an attribute's type in the database and an Objective-C class.	Edit the table cell.
External Type	The data type of the attribute as it's understood by the database.	Choose another value from the pull-down list
Width	The maximum width (applies to string and raw data only).	Edit the table cell.
Column	The database name of the column that corresponds to the attribute.	Edit the table cell.
Definition	The definition for a derived attribute.	Edit the table cell.
Allows Null	Indicates whether the attribute can have a NULL value.	Click in the table cell to toggle the check on and off.
Scale	The number of digits to the right of the decimal point. Can be negative.	Edit the table cell.
Precision	The number of significant digits.	Edit the table cell.
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.
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.
Value Type	The format type for custom value classes such as "TIFF" or "RTF." This type name is used with the EODatabaseCustomValues protocol to identify data formats for custom values.	Edit the table cell.

Using Custom Data Types

Some attributes, such as TalentPhoto's photo attribute, have custom data types. When you use a custom data type, you are responsible for specifying how the data is read from and written to the database.

To specify a custom data type:

1. Select the attribute for which you want to specify a custom data type and choose Tools m Inspector.



Name: photo
Column: PHOTO
External Type: LONG RAW
Internal Data Type: Custom
External Width:
Class: NSImage
Factory Method: imageWithData:
Conversion Method: TIFFRepresentation
Init Argument: NSData

Figure 16. *Specifying Custom Data Types*

2. Use the pop-up list at the top of the Internal Data Type group to set the data type to Custom.

3. If relevant, specify an external width for your data type.

   BLOB types such as images are usually stored in columns that don't have width constraints.

4. In the Class field, specify the class of your custom data type.
5. In the Factory Method field, specify the class method that will be used to create instances of your class from raw data.

   The arguments for this method should match the type specified in the Init Argument pop-up list.
6. In the Conversion Method field, specify the method that will be used to convert your data into a form that can be stored in the database.

   This method should return an NSData object if the Init Argument type is NSData or Bytes, otherwise it should return an NSString.
7. Use the Init Argument pop-up list to indicate the data type (NSData, NString) with which your custom objects will be initialized, or Bytes if your objects are initialized from raw bytes.

Forming a Relationship

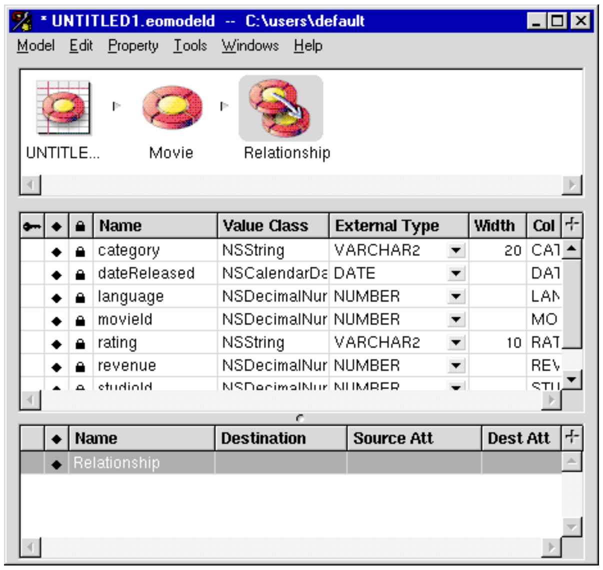
If the database on which your model is based includes definitions for foreign keys, these definitions will automatically be expressed in your model as ready-made relationships.

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 Role and Movie entities.

To form a relationship:

1. Select a source entity in the Model Editor, such as Movie, and navigate to its attributes by double-clicking its icon.
2. Choose Property m Add Relationship.



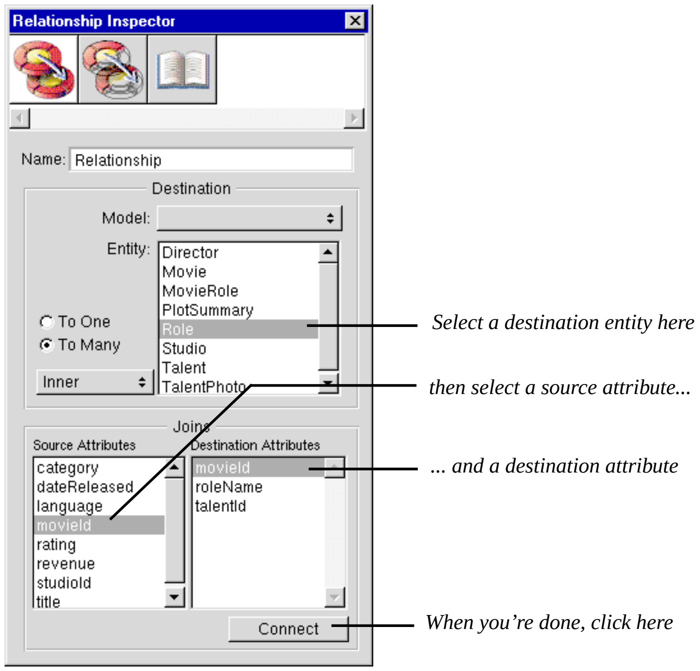
🔑	◆	🔒	Name	Value Class	External Type	Width	Col
	◆	🔒	category	NSString	VARCHAR2	20	CAT
	◆	🔒	dateReleased	NSCalendarDa	DATE		DAT
	◆	🔒	language	NSDecimalNur	NUMBER		LAN
	◆	🔒	movieId	NSDecimalNur	NUMBER		MO
	◆	🔒	rating	NSString	VARCHAR2	10	RAT
	◆	🔒	revenue	NSDecimalNur	NUMBER		REV
	◆	🔒	studioId	NSDecimalNur	NUMBER		STU

	◆	Name	Destination	Source Att	Dest Att
	◆	Relationship			

Figure 17. *Adding a Relationship*

The text “Relationship” appears in the relationship table view at the bottom of the window.

3. Choose Tools m Inspector to display the Relationship Inspector, as shown in Figure 18.


Relationship Inspector
Select a destination entity here
then select a source attribute...
... and a destination attribute
When you're done, click here
Connect

Name: Relationship

Destination

Model:

Entity: Director, Movie, MovieRole, PlotSummary, [✓ Role], Studio, Talent, TalentPhoto

To One [✓ To Many]

Inner

Joins

Source Attributes: category, dateReleased, language, [✓ movieId], rating, revenue, studioId, title

Destination Attributes: [✓ movieId], roleName, talentId

Figure 18. *The Relationship Inspector*

A relationship can be compound, meaning that it can consist of multiple pairs of connected attributes.

You use the Relationship Inspector to specify information about the 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 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.

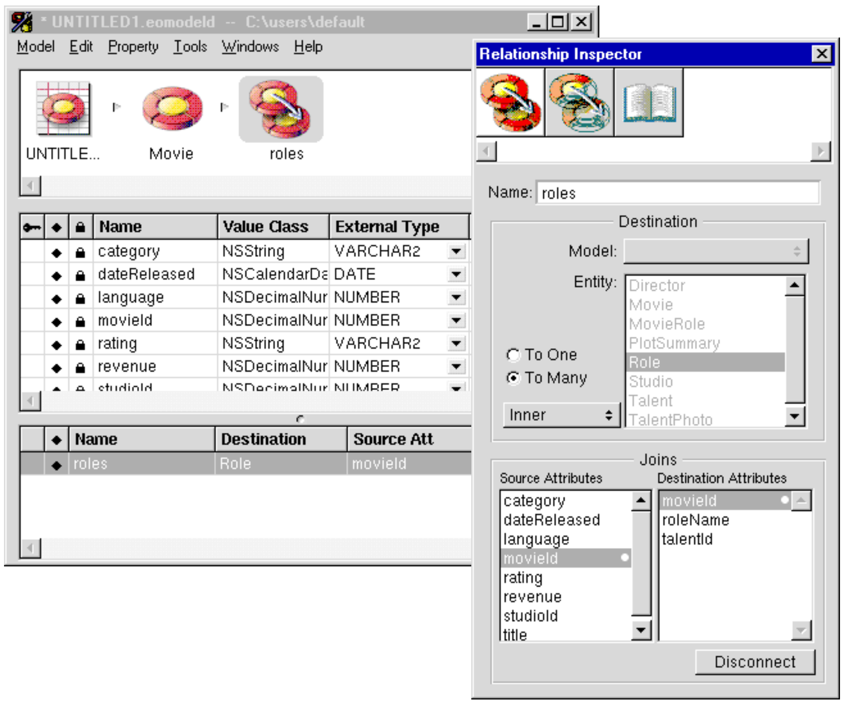
When you use the Relationship Inspector, remember that the settings you define must reflect the corresponding implementation in the database, as well as the features supported by your adaptor. EOModeler doesn't know, for example, if a relationship is to-one or to-many, or if your adaptor supports left outer joins. You need to know your database and your adaptor, and specify relationships accordingly. In addition, to-one relationships must join on the complete primary key of the destination entity as the join component.

4. In the Inspector, select the destination entity (Role) 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. For example, movieId is a foreign key for Role, while it's the primary key for Movie.

5. Select the source attribute (movieId) in the Source Attributes browser.
6. Select the destination attribute (movieId) in the Destination Attributes browser, and click Connect.
7. Make sure the relationship has the proper cardinality (in this example it should be set to To Many since a movie has many roles).
8. By editing in either the Inspector or the relationship table view, give the relationship a name, such as "roles."

   Figure 19 shows what the Model Editor and the Relationship Inspector look like when you get done specifying the relationship.


* UNTITLED1.eomodeld -- C:\users\default
Model Edit Property Tools Windows Help
UNTITLE...
Movie
roles
Relationship Inspector
Destination
Joins
Disconnect

	◆	🔒	Name	Value Class	External Type
	◆	🔒	category	NSString	VARCHAR2
	◆	🔒	dateReleased	NSCalendarDa	DATE
	◆	🔒	language	NSDecimalNur	NUMBER
	◆	🔒	movieId	NSDecimalNur	NUMBER
	◆	🔒	rating	NSString	VARCHAR2
	◆	🔒	revenue	NSDecimalNur	NUMBER
	◆	🔒	studioId	NSDecimalNur	NUMBER

◆	Name	Destination	Source Att
◆	roles	Role	movieId

Name: roles

Model:

Entity: Director, Movie, MovieRole, PlotSummary, Role (selected), Studio, Talent, TalentPhoto

To One [✓ To Many]

Inner

Source Attributes	Destination Attributes
category	movieId
dateReleased	roleName
language	talentId
movieId	
rating	
revenue	
studioId	
title	

Figure 19. *Specifying the Relationship*

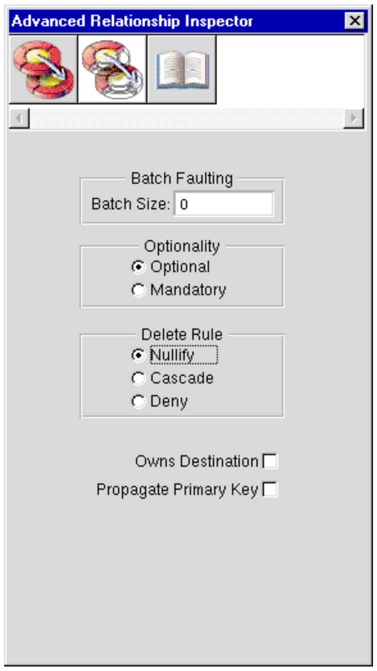
Adding Referential Integrity Rules

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 at the top of the Inspector.

   This displays the Advanced Relationship Inspector.



Batch Faulting

Batch Size: 0

Optionality

[✓ Optional]

Mandatory

Delete Rule

[✓ Nullify]

Cascade

Deny

Owns Destination ☐

Propagate Primary Key ☐

Figure 20. *Advanced Relationship Inspector*

You can use the fields in the Advanced Relationship Inspector to further specify a relationship.

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 that every employee 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 pointer the employee has to the department (nullify)
- Delete the department and all of the employees it contains (cascade)
- Refuse the deletion if the department contains employees (deny)

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.

Adding Derived and Flattened Attributes

The Enterprise Objects Framework supports three different kinds of attributes: simple, derived, and flattened. A simple attribute corresponds to a single column in the root table of the entity, and may be read or updated directly from or to the database.

A derived attribute doesn't map directly to a single column in the root table of the entity. For example, 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.

A flattened attribute is a special kind of derived attribute that you effectively add from one entity to another by traversing a relationship. For example, when you form a to-one relationship between two tables (such as Role and Talent), you can add attributes from the destination entity to the source entity—for example, you can add a lastName attribute to Role 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. Another approach is to define key paths in Interface Builder, as described in the chapter "Creating an Enterprise Objects Framework Project." Key paths allow you to use pointers to traverse the object graph directly, where the most current values of your enterprise objects are maintained. You can also access the values in other objects programmatically, as described in "Designing Enterprise Objects."

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 pointer 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 directly manipulate the object graph 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 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 the chapter "Designing Enterprise Objects").

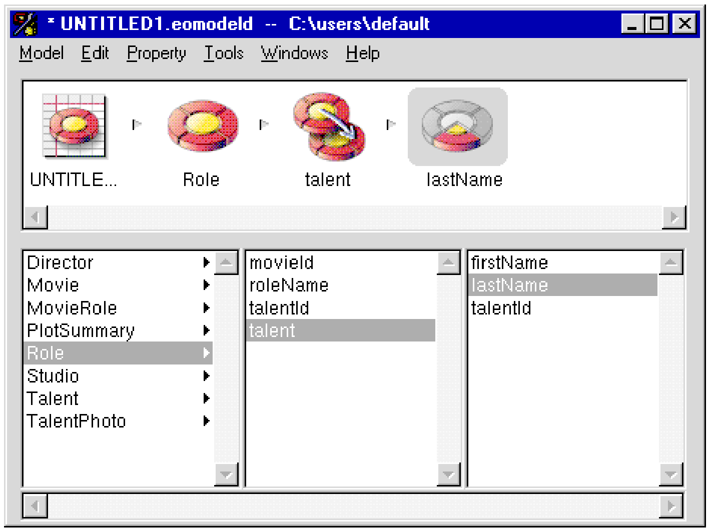
To flatten an attribute, the relationship you traverse must be a to-one relationship.

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 the browser mode, it just makes it easier to see the results of the operation).

   For example, to add the name of an actor to Role, you can add and traverse a talent relationship and add the actor's last name (lastName) to Role as a flattened attribute. Note that 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 m Flatten Property.


* UNTITLED1.eomodeld -- C:\users\default
Model Edit Property Tools Windows Help
UNTITLE...
Role
talent
lastName

Director
Movie
MovieRole
PlotSummary
Role
Studio
Talent
TalentPhoto

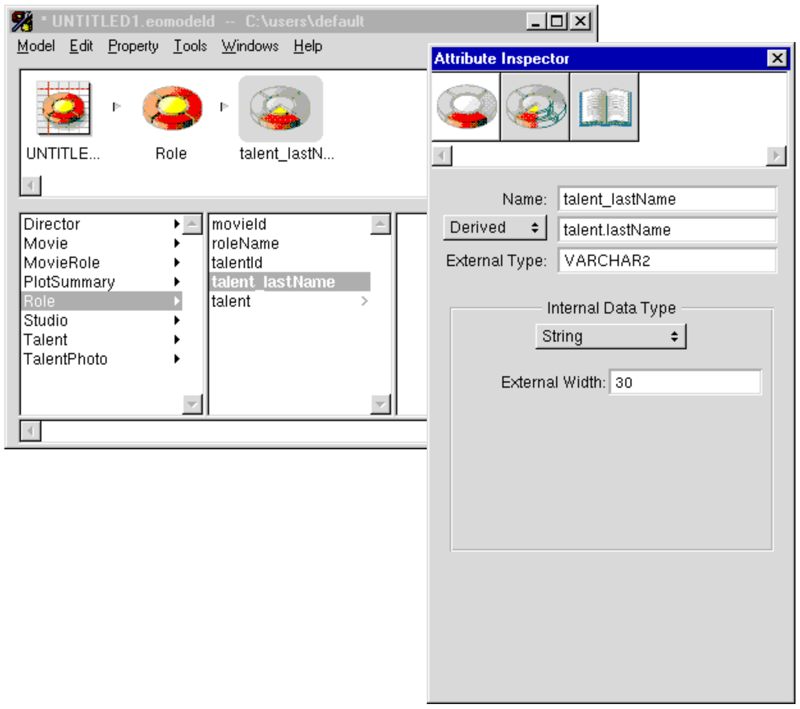
movieId
roleName
talentId
talent

firstName
lastName
talentId

Figure 21. *Adding a Flattened Attribute*

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

3. Choose Tools m Inspector to examine the derived attribute (**talent_lastName**) in the Inspector.



Name: talent_lastName
Derived: talent.lastName
External Type: VARCHAR2
Internal Data Type: String
External Width: 30

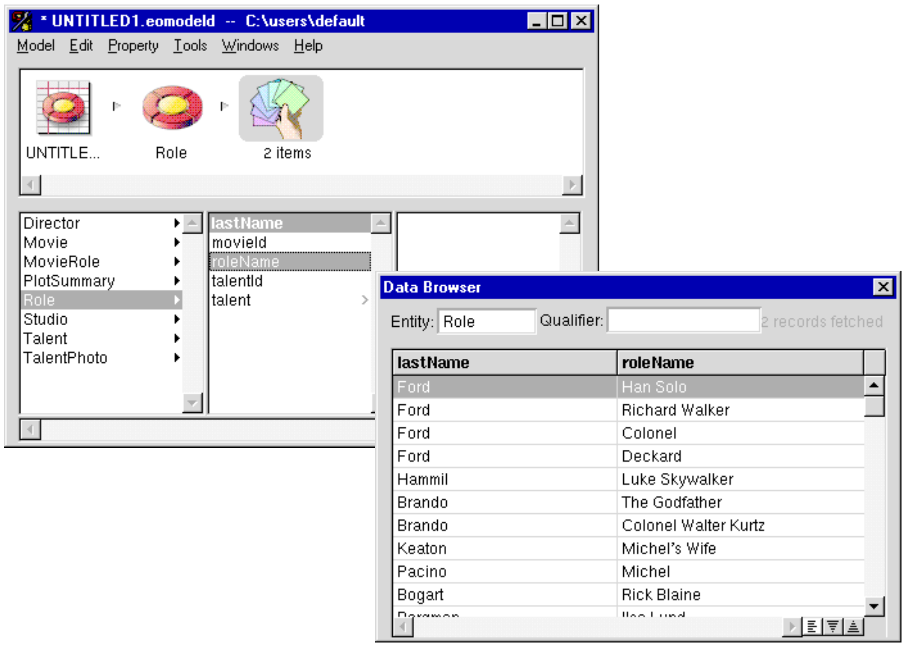
Figure 22. *Examining a Flattened Attribute in the Attribute Inspector*

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 the result of creating this flattened attribute, drag a selection of the Role entity's attributes into the Data Browser, as shown in Figure 23. Notice that the Browser includes a column for the flattened attribute **lastName**.



Director
Movie
MovieRole
PlotSummary
Role
Studio
Talent
TalentPhoto

lastName
movieId
roleName
talentId
talent

Entity: Role Qualifier:

lastName	roleName
Ford	Han Solo
Ford	Richard Walker
Ford	Colonel
Ford	Deckard
Hammil	Luke Skywalker
Brando	The Godfather
Brando	Colonel Walter Kurtz
Keaton	Michel's Wife
Pacino	Michel
Bogart	Rick Blaine

Figure 23. *Using the Data Browser to Check Your Model*

Note: To select multiple, non-contiguous attributes in the Model Editor, hold down the Control key while you mouse-click on each attribute.

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.

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 m Add Attribute.

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. 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. But this is just being used for the purpose of an example.

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

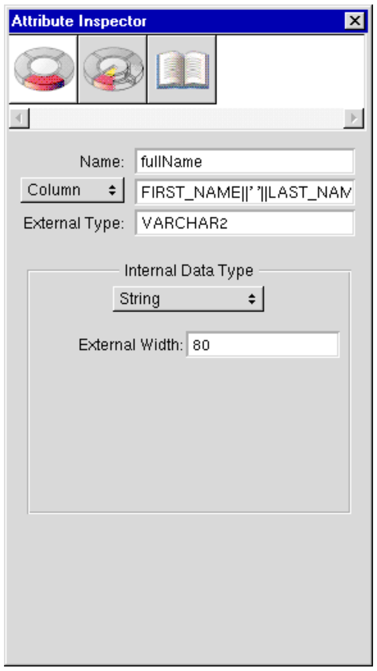
5. Edit the Definition field to supply the SQL needed to specify the derived attribute.

   For example, to concatenate the firstName and lastName attributes in Oracle, type the text FIRST_NAME||' '||LAST_NAME (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 values).

Figure 24 shows the Attribute Inspector with the new attribute **fullName** specified.



Name: fullName

Column: FIRST_NAME||' '||LAST_NAM

External Type: VARCHAR2

Internal Data Type: String

External Width: 80

Figure 24. *Adding an 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.

Adding 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 114, 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.

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 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 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.

   In this example, from Movie select the relationship **directors** to display the properties of Director.

4. In the list of properties for *entity_2*, select the relationship (directors) you want to flatten and choose Property m Flatten Property.

The flattened relationship (in this example, directors_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 directors relationship.

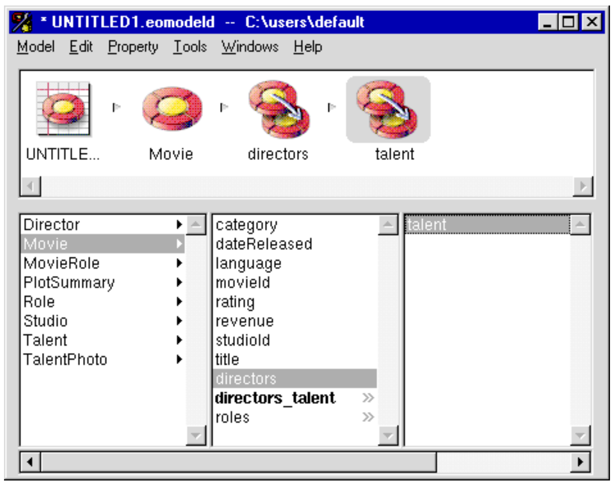


Figure 25. *Flattening a Relationship*

Working With Entities

Once you've refined your model, you're ready to specify enterprise object classes for the entities in the model. There are two stages to specifying an enterprise object class in EOModeler:

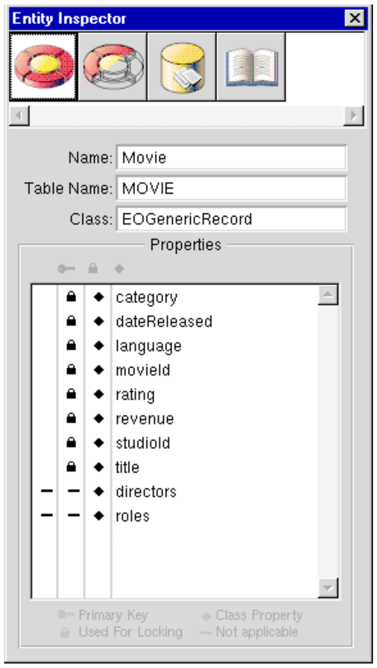
- Using the Model Editor or the Entity Inspector to set the characteristics of an entity and define the mapping between the entity and your enterprise object class.
- Optionally, generating template source files for the enterprise object classes you specify.

Inspecting an Entity

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 a lot of the same tasks using the table mode of the Model Editor, but this section just focuses on the Entity Inspector.

To inspect an entity, select the entity and choose Tools m Inspector.

Figure 26 shows the Entity Inspector for the Movie entity.



Name: Movie
Table Name: MOVIE
Class: EOGenericRecord

Properties

	🔒	◆	category
	🔒	◆	dateReleased
	🔒	◆	language
	🔒	◆	movieId
	🔒	◆	rating
	🔒	◆	revenue
	🔒	◆	studioId
	🔒	◆	title
—	—	◆	directors
—	—	◆	roles

Primary Key
◆ Class Property
🔒 Used For Locking
— Not applicable

Figure 26. *The Entity Inspector*

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 are sure you can predict the result.

Class

The Class field initially contains the text "EOGenericRecord". This is because the default enterprise object class is an EOGenericRecord.

An EOGenericRecord:

- Knows the entity on which it is based.
- Carries its properties as an NSDictionary.
- Implements the EOKeyValueCoding protocol.

To specify a custom class, type the name of the class in this field. For more information on EOKeyValueCoding and creating custom classes, see "Specifying an Enterprise Object Class" on page 124.

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 create 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.

Adding a primary key to 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.

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. Note that 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.

● 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 status flag), you should remove its Class Property icon; note that generated template source files won't include instance variable declarations for these attributes—you'll have to type those in by hand. 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.

Only properties you include in the class will be sent to the enterprise object through key-value coding. Relationships you include as class properties will have EOFaults created for them.

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 corresponding database rows.

To specify the enterprise object class for an entity:

1. Determine the properties from the entity that you want to include in your enterprise object class; every property you want to include should have a corresponding Class Property icon set for it.
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. Note that in all cases, an enterprise object class must conform to the informal protocol EOKeyValueCoding, which specifies methods for accessing values associated with keys ("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 do any one of the following, 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. Generic records implement the key-value coding methods takeValue:ForKey: and valueForKey:. Use EOGenericRecord when you don't need to define special behavior for your class.

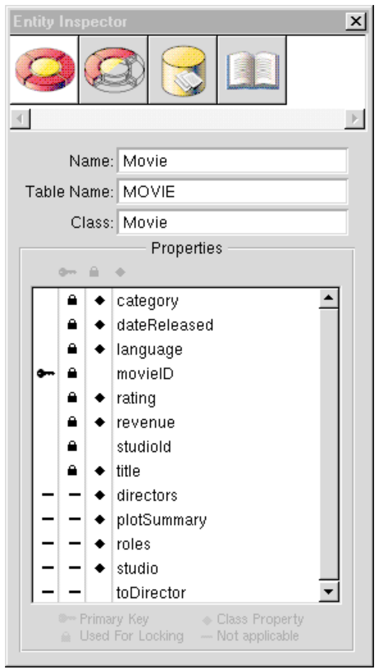
  To use EOGenericRecord, simply leave the text "EOGenericRecord" in the Class field in the Entity Inspector.

- 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.

If you generate template source files for your class, the resulting header and implementation files include definitions of instance variables and accessor methods that can be used by key-value coding. See “Generating Template Source Code Files” on page 126.

For more information on key-value coding and implementing enterprise object classes, see the chapter “Designing Enterprise Objects.”

Figure 27 shows the Model Editor and the Entity Inspector after the primary key and properties have been set.



Name: Movie
Table Name: MOVIE
Class: Movie

Properties

🔑	🔒	◆	
	🔒	◆	category
	🔒	◆	dateReleased
	🔒	◆	language
🔑	🔒		movieID
	🔒	◆	rating
	🔒	◆	revenue
	🔒		studioId
	🔒	◆	title
—	—	◆	directors
—	—	◆	plotSummary
—	—	◆	roles
—	—	◆	studio
—	—		toDirector

🔑 Primary Key ◆ Class Property
🔒 Used For Locking — Not applicable

Figure 27. *Specifying a Class for an Entity*

In Figure 27, note that:

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

Generating Template Source Code Files

Once you finish specifying an enterprise object class, you can generate template source code 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 in Interface Builder. Once you've created a project using Project Builder and included a model file in it, you can generate your template source files and include them directly into the project. For more information on using Project Builder and Interface Builder, see the chapter "Creating an Enterprise Objects Framework Project."

Generating template files produces:

- A header (**.h**) file that declares instance variables for all of the class properties you specified in the Inspector, and accessor methods for those instance variables.

  In the header file, instance variables that correspond to attributes are declared with the type that was specified for them in the Attribute Inspector. This can be an NSString, an NSCalendarDate, an NSNumber, an NSDecimalNumber, or a custom data type. Instance variables that represent to-one relationships are declared to be of type id, while instance variables that represent to-many relationships are NSArrays.

- An implementation (**.m**) file that provides basic implementations for the accessor methods.

To generate template source 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 template source files for entities for which you have specified a custom enterprise object class. In other words, you can't generate template files for EOGenericRecord.

2. Choose Property m Create Template.

   EOModeler displays a Choose Template Name panel. If you opened the model file from Project Builder, the Choose Template 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 template 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.

The files are generated in the specified location.

For example, suppose you define an enterprise object class Movie. The header file generated for this class would resemble the following:

```
// Movie.h
//

#import <EOControl/EOControl.h>

@class Studio;
@class Talent;

@interface Movie : NSObject
{
    int language;
    NSString *category;
    NSCalendarDate *dateReleased;
    NSString *rating;
    NSString *title;
    NSDecimalNumber *revenue;
    id plotSummary;
    Studio *studio;
    NSMutableArray *directors;
    NSMutableArray *roles;
}

- (void)setLanguage:(int) value;
- (int) language;

- (void)setCategory:(NSString *)value;
- (NSString *)category;

- (void)setDateReleased:(NSCalendarDate *)value;
- (NSCalendarDate *)dateReleased;

- (void)setRating:(NSString *)value;
- (NSString *)rating;

- (void)setTitle:(NSString *)value;
- (NSString *)title;

- (void)setRevenue:(NSDecimalNumber *)value;
- (NSDecimalNumber *)revenue;
- (void)setPlotSummary:(id)value;
- (id)plotSummary;

- (void)setStudio:(Studio *)value;
- (Studio *)studio;

- (NSArray *)directors;
- (void)addToDirectors:(Talent *)object;
- (void)removeFromDirectors:(Talent *)object;

- (NSArray *)roles;
- (void)addToRoles:(id)object;
- (void)removeFromRoles:(id)object;


@end
```

Note that:

- Instance variables are declared to be of the type specified in the model. For example, revenue is declared as an NSDecimalNumber and dateReleased is declared as an NSCalendarDate. Instance variables that represent relationships (such as directors) are NSMutableArrays.
- The implementation (.m) file includes an implementation for each of the accessor methods. 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; }
```

Customizing Template Generation

When you create a project with the type “EOF Application,” it inserts two files into the project’s Supporting Files suitcase: **EOInterfaceFile.template** and **EOImplementationFile.template**. You can use these files to customize your **.h** and **.m** file output, respectively. In their unmodified form these files match the template generation scheme used by EOModeler.

Setting Other Information for an Entity

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

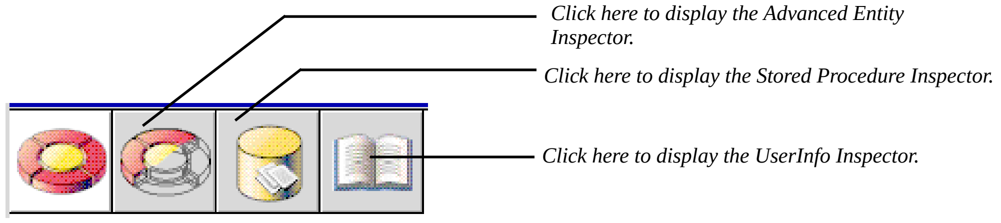
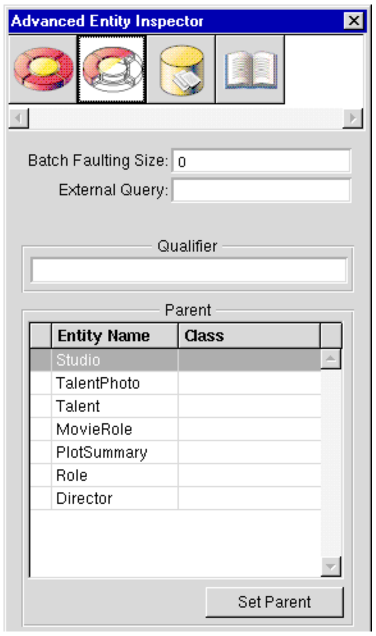

Click here to display the Advanced Entity Inspector.
Click here to display the Stored Procedure Inspector.
Click here to display the UserInfo Inspector.

Figure 28. *Icons for Navigating to Other Inspectors*

Advanced Entity Inspector

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

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



Batch Faulting Size: 0
External Query:

Entity Name	Class
Studio	
TalentPhoto	
Talent	
MovieRole	
PlotSummary	
Role	
Director	

Figure 29. *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.

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). 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 "Designing Enterprise Objects."

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).

Read Only

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

Is Abstract Entity

The Is Abstract Entity 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 example 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.

Most of the features in the Advanced Entity Inspector relate to inheritance. EOModeler also lets you add a new entity as a subclass of the selected entity. To do this, select the entity you want to use as the parent and choose Property m Create Subclass. A new entity is created that maps to the same database table as the parent entity.

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, as described in "Working With Stored Procedures" on page 132.

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 could put information in the UserInfo dictionary to be used by delegate methods that perform operations on the entity.

Working With Stored Procedures

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 in EOModeler.

To add a stored procedure, select the model icon in the Model Editor and choose Property m Add Stored Procedure. You can then edit the stored procedure in the stored procedures view.

To display the stored procedures view, choose Tools m Stored Procedures.

To add arguments for a stored procedure, display the Stored Procedures view and choose Property m Add Argument.

Once stored procedures have been added to your model, you can use the Stored Procedures Inspector to specify stored procedures that should be executed for an entity when a particular database operation (such as insert or delete) occurs (as described in “Stored Procedures Inspector” on page 131).

Working with Multiple Models and Databases

The entities in one model can have relationships to the entities in another model that maps to a different database.

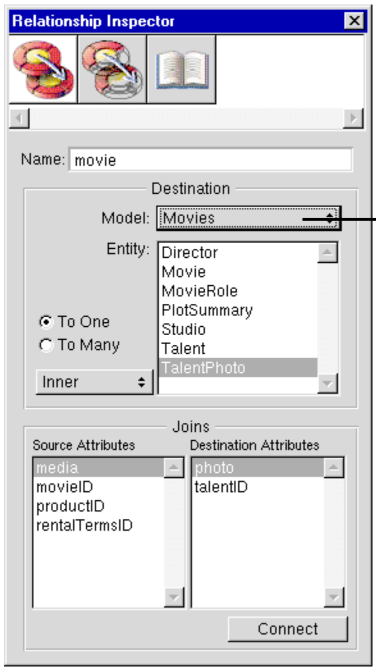
When you add a model to a project, it becomes part of an EOModelGroup, even if the model group only contains that one model (for more information on model groups, see the EOModelGroup class specification). Each subsequent model that you add to the project automatically becomes part of the EOModelGroup. Entity names must be unique within a single EOModelGroup; you can’t use the same entity name in two different models in the same group.

You can form relationships from one model to other models in the same EOModelGroup. You do this 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 VideoTape entity in the Rentals sample database and the Movie entity in the Movies 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.



Name: movie
Model: Movies
Entity: Director, Movie, MovieRole, PlotSummary, Studio, Talent, TalentPhoto (selected)
[✓ To One] To Many
Inner
Source Attributes: media (selected), movieID, productID, rentalTermsID
Destination Attributes: photo (selected), talentID

Use this pop-up list to choose the model that contains the entity you want to use as the destination of the relationship.

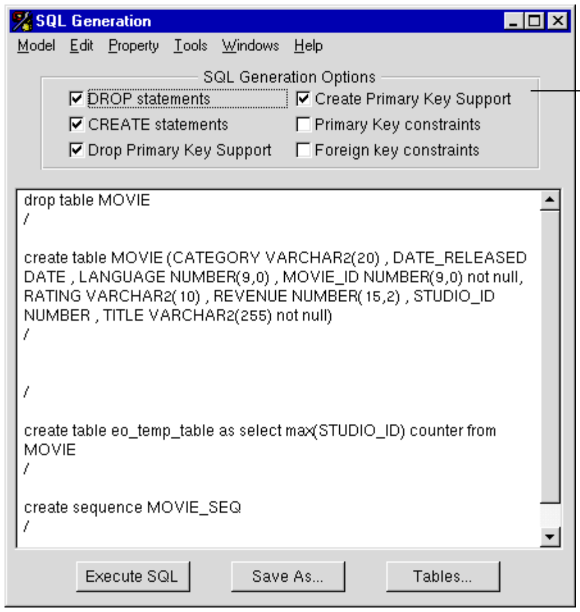
3. Specify the relationship as you normally would.

Note: You can’t flatten properties across databases, nor can you map inheritance hierarchies across databases.

Generating Schema

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 m Generate SQL. The SQL Generation panel appears, as shown in Figure 30.



SQL Generation Options:

- [✓] DROP statements
- [✓] CREATE statements
- [✓] Drop Primary Key Support
- [✓] Create Primary Key Support
- [ ] Primary Key constraints
- [ ] Foreign key constraints

```
drop table MOVIE
/

create table MOVIE (CATEGORY VARCHAR2(20) , DATE_RELEASED
DATE , LANGUAGE NUMBER(9,0) , MOVIE_ID NUMBER(9,0) not null,
RATING VARCHAR2(10) , REVENUE NUMBER(15,2) , STUDIO_ID
NUMBER , TITLE VARCHAR2(255) not null)
/



/

create table eo_temp_table as select max(STUDIO_ID) counter from
MOVIE
/

create sequence MOVIE_SEQ
/
```

Click these options to dynamically change the contents of the generated SQL.

Figure 30. *Generating SQL*

Setting Adaptor Information

In addition to describing modeling objects, 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 m 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.

Switching Adaptors

You can also change the database your model is based on by choosing Model m Switch Adaptor. This displays the New Model panel, where you can select a different adaptor.

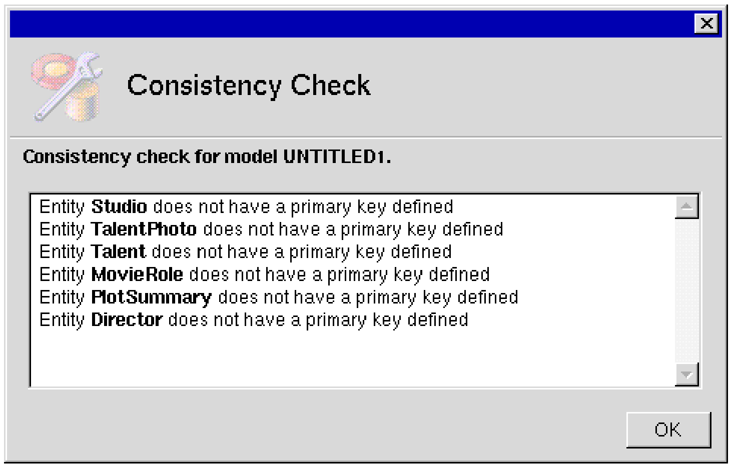
Checking for Consistency

EOModeler provides consistency checking to confirm that your model is valid. A valid model is one in which there are no entities without primary keys, and no relationships without join components. Further, consistency checking is invoked when you attempt to make a change in one part of the model that would invalidate another part of the model (for example, if you try to delete an element that's referenced elsewhere).

You can explicitly check your model at any point by choosing Model m Check Consistency. Consistency checking is also invoked automatically whenever you perform certain operations. These operations and the associated checks that EOModeler performs are described in the following table:

When you attempt to...	EOModeler checks for...
Save the model	Entities without primary keys Relationships without join components
Delete an entity	References other entities may have to any aspect of this entity (for example, to its attributes)
Delete an attribute	References to this attribute anywhere else in the model
Delete a relationship	References to this relationship anywhere else in the model
Change relationship cardinality	References to this relationship anywhere else in the 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 31.



Consistency check for model UNTITLED1.

Entity **Studio** does not have a primary key defined
Entity **TalentPhoto** does not have a primary key defined
Entity **Talent** does not have a primary key defined
Entity **MovieRole** does not have a primary key defined
Entity **PlotSummary** does not have a primary key defined
Entity **Director** does not have a primary key defined

Figure 31. *Checking for Consistency*

By default, consistency checking is performed whenever you save a model file. You can change this behavior with the Preferences panel.

Saving the Model

To save your model, choose Model m Save. If you’re planning to use your model in application for which you’ve already created a project, save the model into your project folder. You will be prompted to add it to the project; click OK.