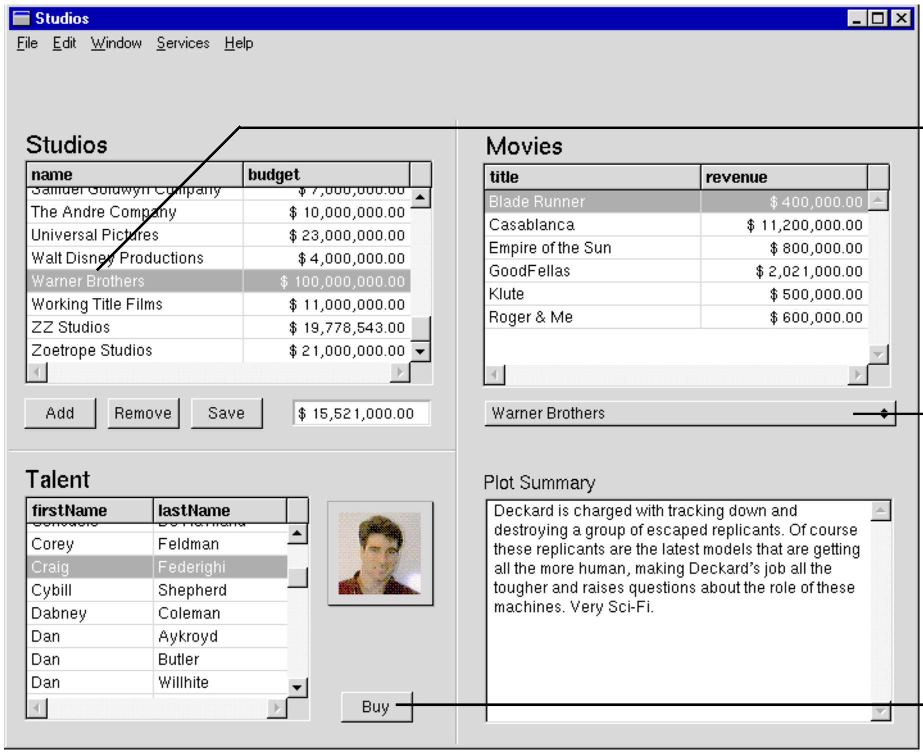
Chapter 1

Getting Started

This chapter introduces you to Enterprise Objects Framework by showing you how to create a simple application. The steps you take in creating this application demonstrate the principles you'll use in every other application you develop.

The application you'll be creating in this chapter, Studios, is based on the Movies sample database distributed with Enterprise Objects Framework (you must have the sample databases installed to do this tutorial). It centers around three enterprise objects: Studio, Movie, and Talent. Studios own movies, and they have a budget for buying new movies. Movies feature actors, or "talent." The Studios application lets you transfer movies between studios and buy all of the movies starring a particular actor. It also lets you add, modify, and delete studios.



Studios

name	budget
Samuel Goldwyn Company	$ 7,000,000.00
The Andre Company	$ 10,000,000.00
Universal Pictures	$ 23,000,000.00
Walt Disney Productions	$ 4,000,000.00
Warner Brothers	$ 100,000,000.00
Working Title Films	$ 11,000,000.00
ZZ Studios	$ 19,778,543.00
Zoetrope Studios	$ 21,000,000.00

$ 15,521,000.00

Movies

title	revenue
Blade Runner	$ 400,000.00
Casablanca	$ 11,200,000.00
Empire of the Sun	$ 800,000.00
GoodFellas	$ 2,021,000.00
Klute	$ 500,000.00
Roger & Me	$ 600,000.00

Warner Brothers

Talent

firstName	lastName
Corey	Feldman
Craig	Federighi
Cybill	Shepherd
Dabney	Coleman
Dan	Aykroyd
Dan	Butler
Dan	Willhite

Plot Summary

Deckard is charged with tracking down and destroying a group of escaped replicants. Of course these replicants are the latest models that are getting all the more human, making Deckard's job all the tougher and raises questions about the role of these machines. Very Sci-Fi.

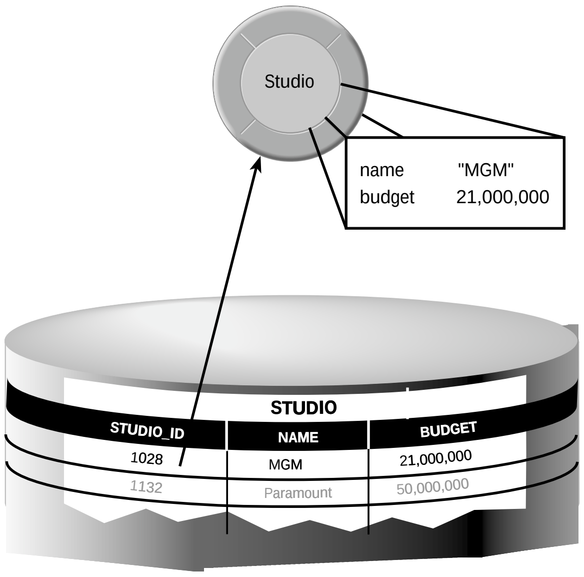
Select a studio to display its movies.

Use the pop-up list to transfer a movie to a different studio.

Click here to transfer all of the movies starring the selected actor to the selected studio.

Enterprise Objects and Relational Databases

The Studio, Movie, and Talent enterprise objects correspond to tables in a relational database. For example, the Studio enterprise object corresponds to the STUDIO table in the database, which has NAME and BUDGET columns. The Studio enterprise object class in turn has **name** and **budget** instance variables, or *class properties* (instance variables based on database data are called "class properties"). In an application, Studio objects are instantiated using the data from a corresponding database row, as shown in the following figure:



Enterprise objects are instantiated from a corresponding database row. They add behavior to the data they contain.

The enterprise objects in your application are not just a static representation of your database data, however. Enterprise objects add behavior to your data. For example, the Studio enterprise object class has a method for calculating the studio's portfolio value based on the revenue of its movies. It also has a method for buying all of the movies starring a specified actor.

Enterprise Objects Framework manages the interaction between the database, your enterprise objects, and the user interface. Its primary responsibilities are as follows:

- Fetching data from relational databases into enterprise objects
- Binding data in enterprise objects to the user interface
- Keeping objects in the application in sync with each other, with the database, and with the user interface

For Mach Users Only

The examples in this chapter are based on Windows NT. Most of the operations you perform on Mach are the same, but there are a few slight differences:

- On Mach the EOModeler and Project Builder applications are located in the **/NextDeveloper/Apps** directory.
- Some of application behavior may be subtly different: For example, on Windows NT, EOModeler starts by displaying an empty model. On Mach it just displays the menu.
- After you build an application on Windows NT, you run it by opening the ***<AppName>*.app** directory in your project and double-clicking ***<AppName>*.exe**. On

Creating the Studios Application

As with most Enterprise Objects Framework applications, you create the Studios application using the following ingredients:

- **A model you produce using the EOModeler application provided with Enterprise Objects Framework.** A model defines a mapping between your enterprise objects and data in a relational database.
- **A user interface.** You can either use Interface Builder to construct a conventional user interface, or WebObjects Builder if you want to build an application that can be deployed on the World Wide Web. The examples in this tutorial use Interface Builder.
- **Source code for enterprise object classes.** In the Studios application, these are Studio and Talent. Movie uses the default enterprise object class, EOGenericRecord, since it has no custom behavior. This is described in more detail in later sections.

In addition, the Studios application requires a database server on which you've installed the Movies example database. The final ingredients in the application are the Enterprise Objects Framework classes and protocols, which you link into your application.

In this tutorial you'll learn the basic things you must do to create an Enterprise Objects Framework application. You'll discover how to:

- Create a new project using Project Builder.

- Create a new model based on the Movies database using EOModeler.
- Edit your project's nib file in Interface Builder.
- Write source code for the Studio and Movie enterprise object classes.
- Build your project in Project Builder.

What is an Enterprise Object?

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

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

The ingredients that make up an enterprise object are its class definition and the data values from the database row or record with which the object is instantiated. An enterprise object also has a corresponding model that defines the mapping between the class' object model and the database schema.

Creating the Studios Project

Every Enterprise Objects Framework application starts out as a *project*. A project is a repository for all the elements that go into the application, such as source code files, makefiles, frameworks, libraries, the application's user interface, sounds, and images. You use the Project Builder application to create and manage projects.

1 **Start Project Builder.**

In the OPENSTEP Enterprise program group, double-click the Project Builder icon.

You must create or open a project to get Project Builder's main window. The New Project panel allows you to specify a new project's name and location.

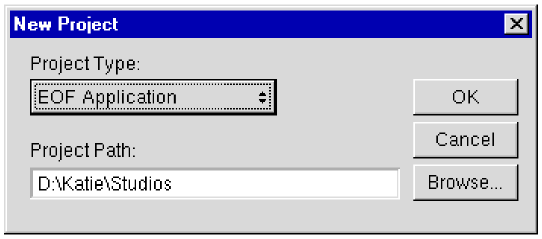
2 **Make a new project.**

Choose Project m New.

In the New Project panel, you can either use the Browse... button to navigate to the directory in which you want to put the new project, or you can type the full path. Give the project the name "Studios."

Use the Project Type pop-up list to set the project type to EOF Application. This adds all of the necessary frameworks to your project.

Click OK to create the project.



Project Type: EOF Application

Project Path: D:\Katie\Studios

Project Builder creates a project directory named after the project—in this case Studios—and populates this directory with an assortment of ready-made files and directories. It then displays its main window.

When you create a project with the type "EOF Application," it automatically adds all of the frameworks you need to your project (EOAccess.framework, EOControl.framework, and EOInterface.framework). A *framework* is a project type that packages a shared dynamic library with its headers, documentation, and resources.

Creating the Model

One of the fundamental features of Enterprise Objects Framework is that it maps the data in relational databases to objects. The correspondence between an enterprise object class and stored data is established and maintained by using a *model*. A model defines, in entity-relationship terms, the mapping between enterprise object classes and a database.

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)
Row	—	Enterprise object instance

In addition to storing a mapping between the database schema and enterprise objects, a model file stores information needed to connect to the database server. This connection information includes the name of an adaptor to load so that Enterprise Objects Framework can communicate with the database.

1 **Launch EOModeler.**

In the OPENSTEP Enterprise program group, double-click the EOModeler icon.

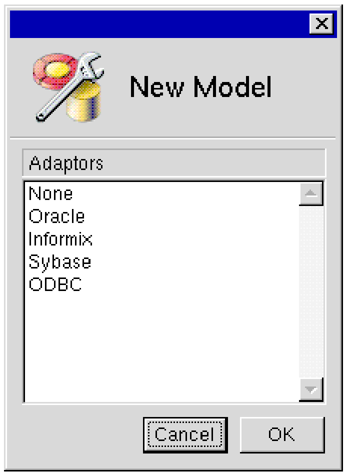
An empty model opens.

2 **Open a new model.**

Choose Model m New.

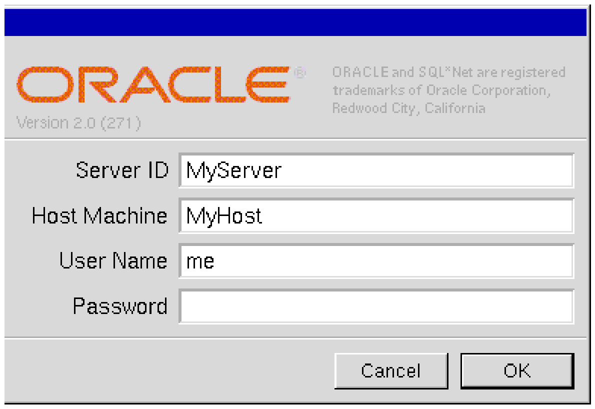
In the New Model panel, select the adaptor for the database you want to use.

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 Movies database included with Enterprise Objects Framework.

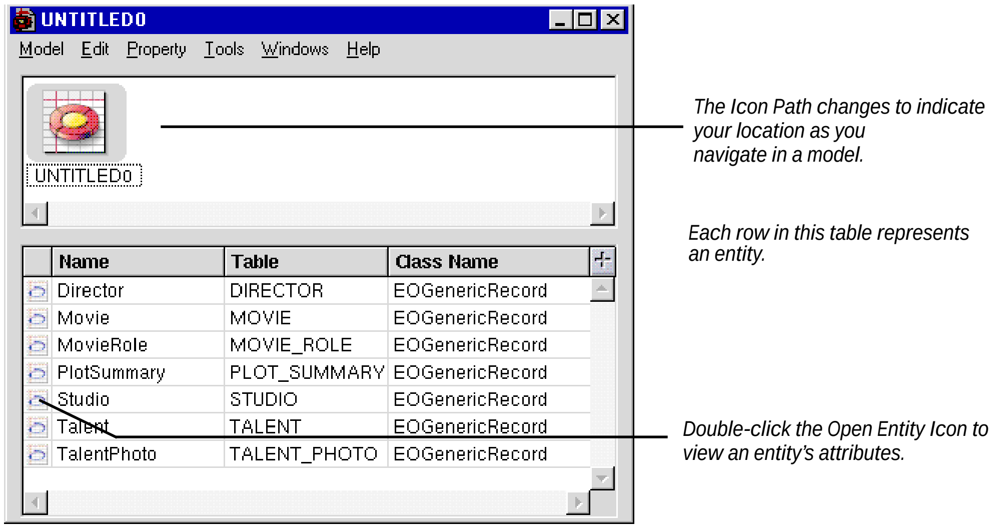
3 **Fill in the login panel.**


ORACLE® Version 2.0 (271)

ORACLE and SQL*Net are registered trademarks of Oracle Corporation, Redwood City, California

Server ID	MyServer
Host Machine	MyHost
User Name	me
Password	

When you first log in to a database, EOModeler uses an adaptor to read the data dictionary from the database and create a default model. This model is displayed in the Model Editor (shown below), which lists the entities available for the database you specified in the login panel.


UNTITLED0
Model Edit Property Tools Windows Help
UNTITLED0

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

The Icon Path changes to indicate your location as you navigate in a model.
Each row in this table represents an entity.
Double-click the Open Entity Icon to view an entity's attributes.

The default model created for you by EOModeler when you open a new model is just a starting point. For most applications, you need to do some additional work to your model to make it useful in your application. To produce a model that can be used in the Studios application, you will ultimately need to do all of the following:

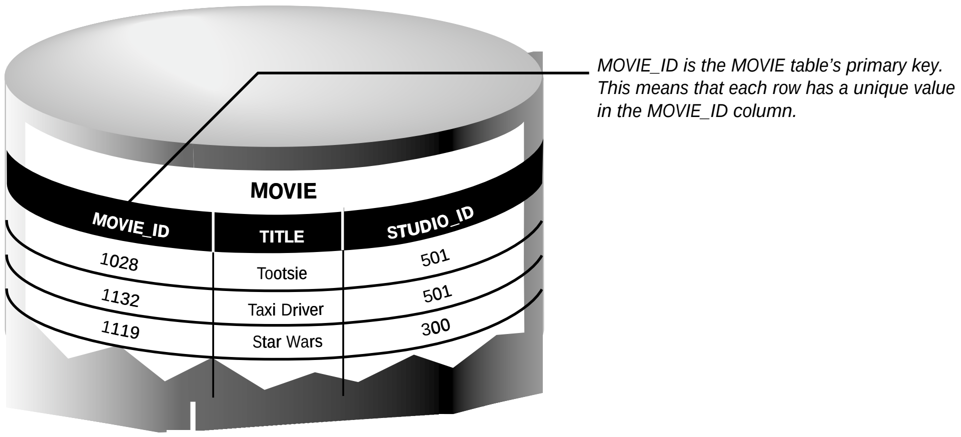
- Assign the primary key for each entity.
- Add relationships to the Studio, Movie, Talent, and Movie Role entities.

- Generate template source code for the Studio and Talent classes.

These steps are described in more detail throughout this tutorial.

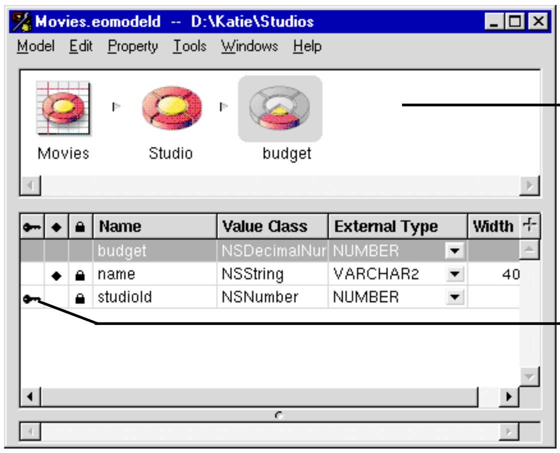
Assigning Primary Keys

In a relational database, each table has a column or combination of columns whose values are guaranteed to uniquely identify each row in that table. For example, in the Movies database the MOVIE table has as its primary key the column MOVIE_ID. Each row in the MOVIE table has a different value in the MOVIE_ID column, which uniquely identifies that row. Two movies could have the same name, but still be distinguished from each other by their primary keys.


MOVIE
MOVIE_ID
TITLE
STUDIO_ID
1028
Tootsie
501
1132
Taxi Driver
501
1119
Star Wars
300
MOVIE_ID is the MOVIE table's primary key. This means that each row has a unique value in the MOVIE_ID column.

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 of your entities in EOModeler.

1 **Assign a primary key to each entity.**


Movies.eomodeld -- D:\Katie\Studios
Model Edit Property Tools Windows Help
Movies
Studio
budget

Name	Value Class	External Type	Width
budget	NSDecimalNur	NUMBER	
name	NSString	VARCHAR2	40
studioId	NSNumber	NUMBER	

Click on icons in the icon path to navigate the model.

Click in the primary key column for the attribute you intend to use as your entity's primary key. The primary key icon means that the attribute is used as the primary key.

The following table lists the primary keys you need to assign to each of the entities in the model. Note that some of the entities (such as Director) have a *compound primary key*; that is, a primary key that is composed of more than one attribute. For more discussion of this subject, see the appendix "Entity-Relationship Modeling."

For the entity...	Assign as primary key attributes...
Director	movieID and talentID
Movie	movieID
MovieRole	movieID and roleName
PlotSummary	movieID
Studio	studioID
Talent	talentID
TalentPhoto	talentID

Before You Go On

Save your model by choosing Model m Save. In the Save panel, give the model the name Movies and save it into your Studios project folder. When you are prompted to add the model to your project, click OK. You'll be returning to

EOModeler to enhance your model in later exercises, but for now you're ready to build the first stage of the Studios application.

Creating the User Interface for the Studios Application

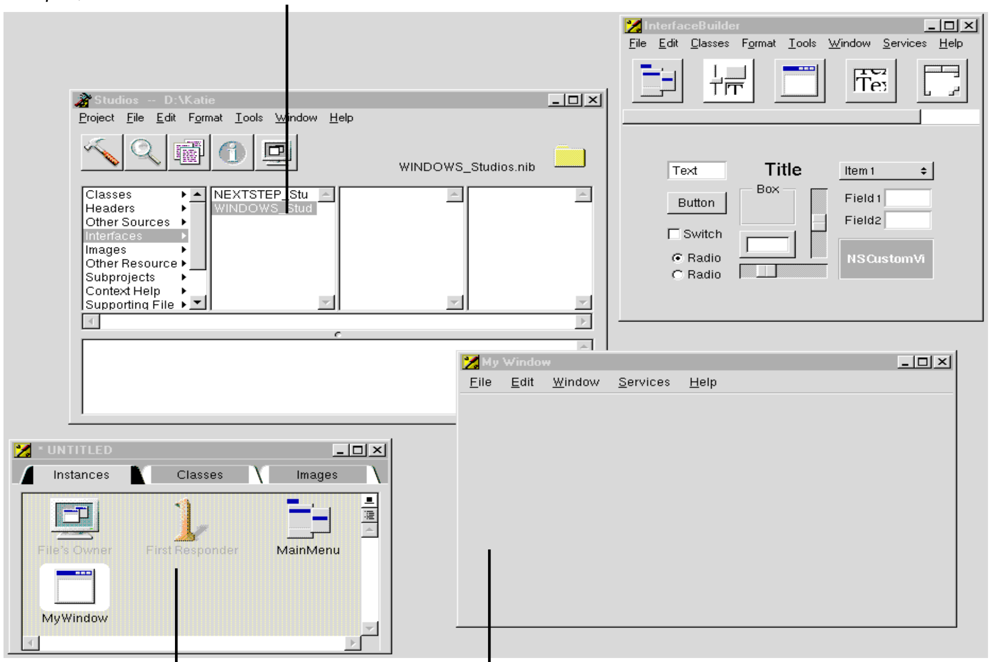
When you create an application project, Project Builder puts the *main nib file* in the Interfaces suitcase. A nib file is primarily a description of a user interface (or part of a user interface). The main nib file contains the main menu and any windows and panels you want to appear when your application starts up.

The default main nib file created by Project Builder is like a blank canvas, ready for you to craft the interface. Look in the Interfaces suitcase for nib files.

1 **Open the main nib file.**

Locate the appropriate nib file in the project browser. On Windows NT this is WINDOWS_Studios.nib; on Mach it is NEXTSTEP_Studios.nib.

To open, double-click the file name.



nib file window

When you first open the application's main nib file, Interface Builder displays a blank window.

By default, the window entitled “My Window” appears when the application is launched.

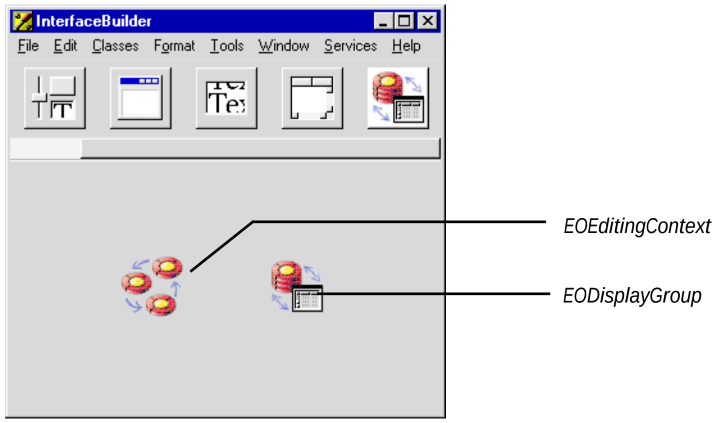
Note: The Interface Builder application is located in the OPENSTEP Enterprise program group. The icon for the application is this:

In Interface Builder you construct a user interface by dragging objects from a palette and dropping them onto the window. The palette provided for use in the Enterprise Objects Framework is the EOPalette. The EOPalette includes two objects: EODisplayGroup and EOEditingContext.

2 **Load the EOPalette.**

Choose Tools m Palettes m Open.

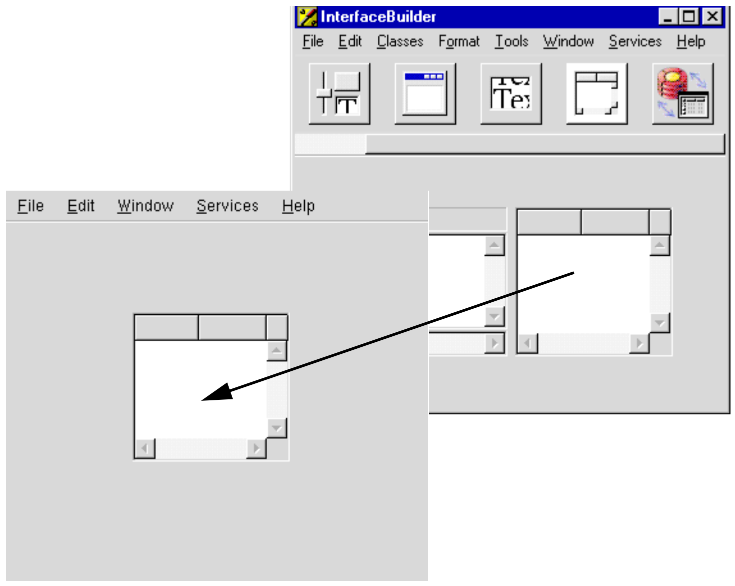
In the Open Palette panel, navigate to **Next/NextDeveloper/Palettes** and double-click **EOPalette.palette**.



3 **Add an NSTableView to the window.**

Select the TabulationViews Palette.

Drag a table view from the palette onto the window.


InterfaceBuilder
File Edit Classes Format Tools Window Services Help
File Edit Window Services Help

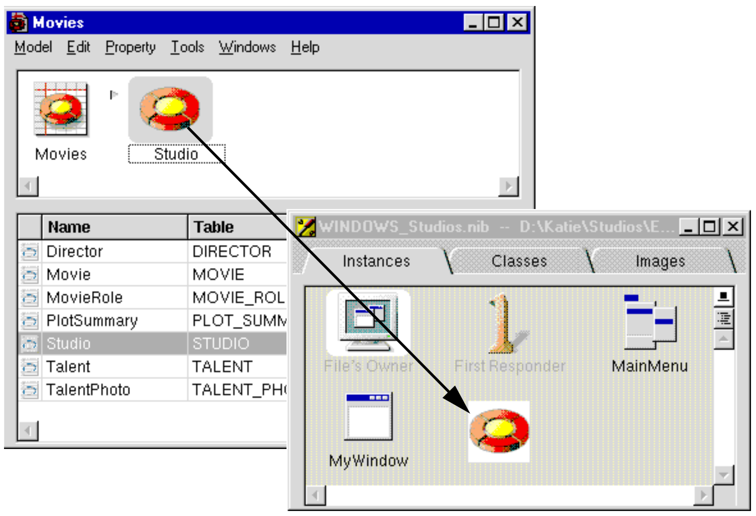
You can use an NSTableView object (on the TabulationViews Palette) to display rows of data in your user interface. A new NSTableView has two columns with enough room for about six records. You can add columns by copying and pasting an existing column.

To display data in your user interface, you need an EODisplayGroup object. *EODisplayGroups* transport values between an enterprise object and a user interface object. You also need an EODatabaseDataSource, which acts on behalf of the EODisplayGroup to fetch enterprise objects from the database. In combination, EODisplayGroup and EODatabaseDataSource coordinate the flow of data between the user interface and the database.

To produce an *entity EODisplayGroup* (which consists of an EODisplayGroup pre-connected to an EODatabaseDataSource), drag an entity from EOModeler into the nib file window.

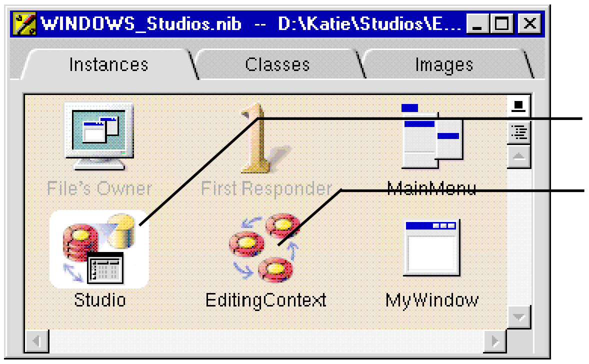
4 **Drag the entities you want to use in your application from EOModeler into the nib file window.**

The model you use must have been added to your project.You usually accomplish this by saving your model into your project folder.



Name	Table
Director	DIRECTOR
Movie	MOVIE
MovieRole	MOVIE_ROL
PlotSummary	PLOT_SUMM
Studio	STUDIO
Talent	TALENT
TalentPhoto	TALENT_PH

The resulting entity EODisplayGroup has the same name as the corresponding entity.



The EODisplayGroup has the same name as the entity from which it was created.

An EOEditingContext object is added to your application along with the first entity you drag into Interface Builder. Because your application typically needs only one EOEditingContext, this object is only added once.

What is an EOEditingContext?

When you drag an entity into the nib file window from your model, an EOEditingContext object is added to your application along with the EODisplayGroup that's created from the entity. An EOEditingContext manages the graph of enterprise objects in your application. The EOEditingContext is responsible for ensuring that all parts of your application stay in sync. When an enterprise object changes, the EOEditingContext broadcasts a notification so that other parts of the application (such as the user interface) can update themselves accordingly. The EOEditingContext also manages undo, and is the object through which you save changes to the database. For more information, see the EOEditingContext class specification in the *Enterprise Objects Framework Reference*.

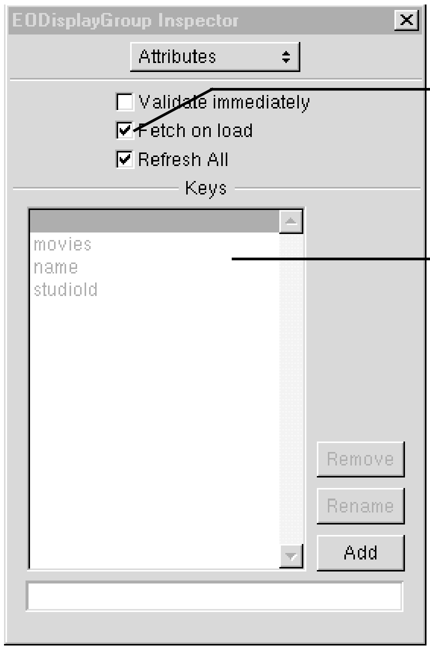
An entity EODisplayGroup has keys that correspond to the properties in its associated enterprise object class. You can examine these keys in the EODisplayGroup Inspector.

5 **Examine the EODisplayGroup in the Inspector.**

Select the Studio EODisplayGroup in the nib file window.

Choose Tools m Inspector.

Make sure that the “Fetch on load” checkbox is checked. This causes data to be fetched from the database when you start your application.



Check here.

The keys listed correspond to the properties you specify for the Studio entity in EOModeler.

Connecting the Studio EODisplayGroup to the Interface

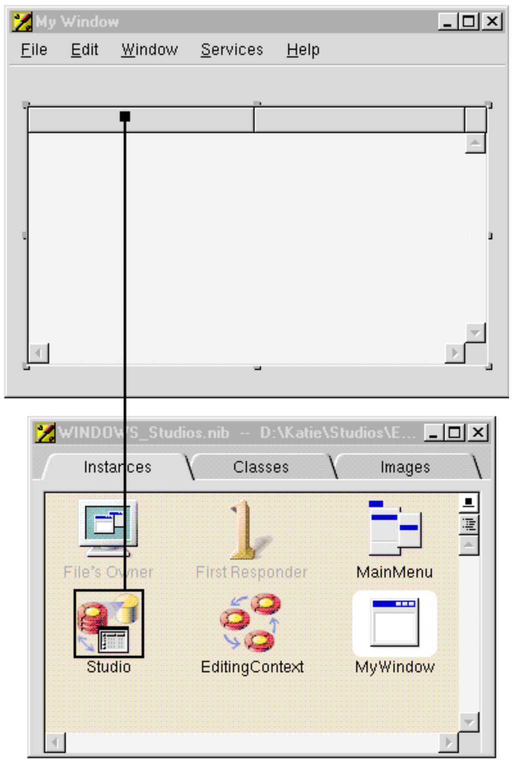
Now that you’ve dragged the Studio entity from EOModeler into your nib file in Interface Builder, you’re ready to create the first stage of the Studios application. You do this by making connections between your table view and the new Studio EODisplayGroup you just added to your nib file.

1 **Form an association between the table view columns and the Studio EODisplayGroup.**

Select the title bar of the column from which you want to connect. To select a title bar, double-click it.

Control-drag a connection line from the column to the Studio EODisplayGroup in the nib file window.

When the Studio EODisplayGroup is outlined in black, release the mouse button.



Interface Builder opens the Connections display of the Inspector panel.

2 **Make the connection.**

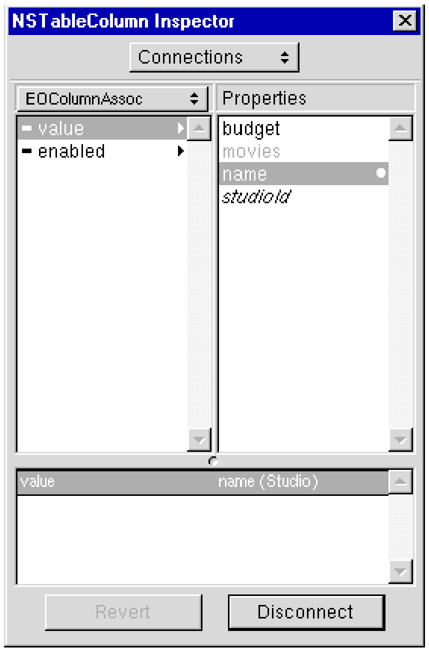
Choose EOColumnAssoc from the pop-up list at the top of the left column.

Select **value** in the left column.

Select **name** in the right column.

Click Connect or double-click **name** to make the connection.

Using the same steps, connect the remaining table view column to the **budget** key.



The Connections Inspector uses different symbols and notations to distinguish different types of aspects and class keys.

Class keys that are italicized aren't class properties.

Class keys that are gray text don't match the type required by the selected aspect, and the Inspector won't let you connect to them.

▬ *This symbol next to an aspect in the left column means that the aspect should be bound to a class key that's based on an attribute (as opposed to one that represents a relationship).*

› *This symbol next to an aspect in the left column means that the aspect should be bound to a class key that represents a to-one relationship.*

» *This symbol next to an aspect in the left column means that the aspect should be bound to a class key that represents a to-many relationship.*

What is an Association?

When you made a connection from the table columns to the Studio EODisplayGroup in the preceding exercise, you formed an *association*.

EODisplayGroups use associations (EOAssociations) to mediate between enterprise objects and the user interface. An association ties a single user interface object, such as a table column, to a key (a named property) in an enterprise object or objects managed by the EODisplayGroup.

Associations keep the user interface synchronized with enterprise object values. When an object changes, its display in the user interface updates to reflect the change. Likewise, when the user edits the user interface, the values in the object are updated accordingly.

Associations can have multiple *aspects*. For example, in the preceding exercise you selected the **value** aspect for the EOColumnAssociation to display all of the class keys whose values you could choose to display in the table column. EOColumnAssociation has one other aspect, **enabled**.

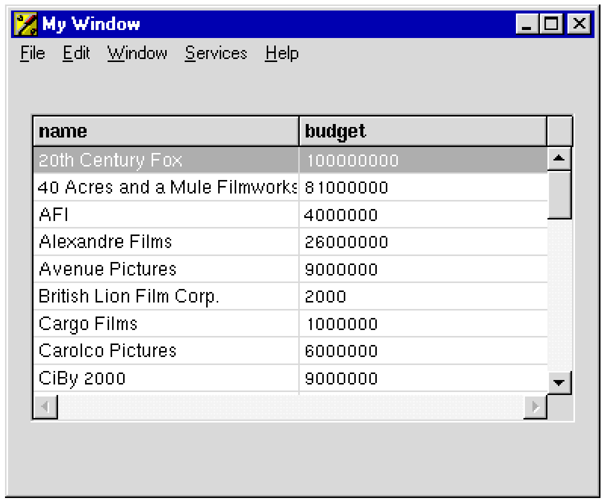
Enterprise Objects Framework includes associations for different types of user interface objects, such as table columns, text fields, pop-up lists, and so on. Each association has multiple aspects.

For a complete discussion of this subject and a listing of all possible associations, see the EOAssociation class and subclass specifications in the *Enterprise Objects Framework Reference*.

Now that you've formed associations for all of the columns in the table view, you can test your interface.

3 **Test the interface.**

Choose File m Test Interface.



name	budget
20th Century Fox	100000000
40 Acres and a Mule Filmworks	81000000
AFI	4000000
Alexandre Films	26000000
Avenue Pictures	9000000
British Lion Film Corp.	2000
Cargo Films	1000000
Carolco Pictures	6000000
CiBy 2000	9000000

Note that because you enabled the "Fetch on load" option for the Studio EODisplayGroup in the Inspector, the data is automatically fetched when you test your interface.

Formatting Currency and Dates

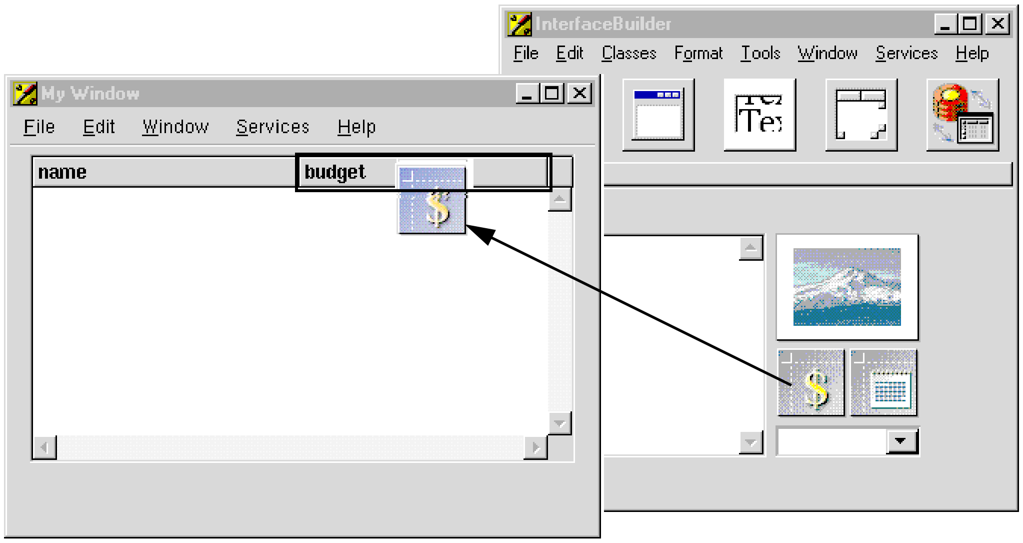
When you test your interface, you may notice that the values listed in the budget table column have no format. That is, the data in the column represents currency, but it has no comma separators or decimal points.

Interface Builder includes two formatters: one for currency, and one for dates. You can use these formatters to specify how a user interface control such as a text field or table view column formats the data it displays. Formatters also ensure that users enter data that is of the correct type and in the correct format.

1 **Add formatting.**

From the palette, drag the Currency formatter onto the **budget** column.

Note that the formatter hasn't been successfully added until you see the column head outlined in black.

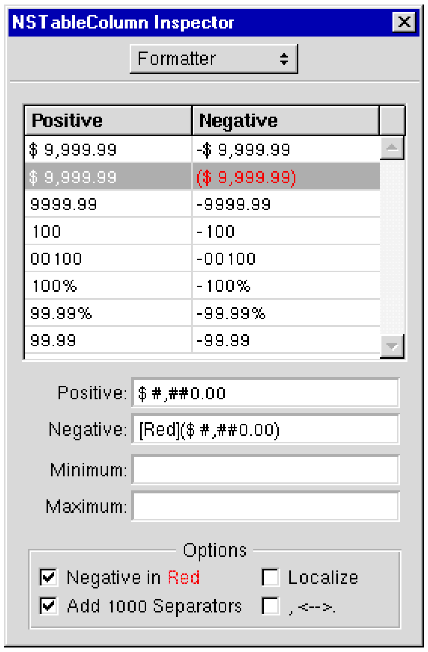

InterfaceBuilder
File Edit Classes Format Tools Window Services Help
My Window
File Edit Window Services Help
name
budget

Once you drag a formatter onto a user interface control, use the Inspector to specify the formatting characteristics.

Display the Inspector for the **budget** table view column.

Use the pop-up list at the top of the Inspector to display the Formatter view.

Select the format you want to use to display the **budget** data.



Positive	Negative
$ 9,999.99	-$ 9,999.99
$ 9,999.99	($ 9,999.99)
9999.99	-9999.99
100	-100
00100	-00100
100%	-100%
99.99%	-99.99%
99.99	-99.99

Positive: $ #,##0.00
Negative: [Red]($ #,##0.00)
Minimum:
Maximum:

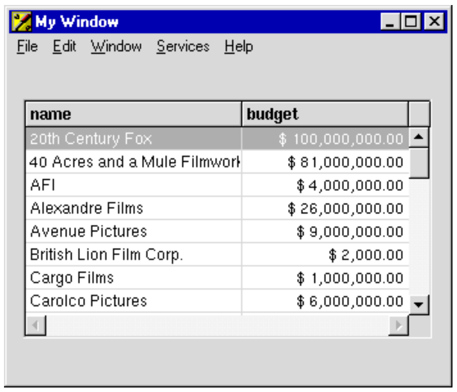
Options
[✓] Negative in Red
[ ] Localize
[✓] Add 1000 Separators
[ ] , <-->.

You can see the effects of the new formatting by testing your interface.

2 **Test your interface.**

Choose File m Test Interface.

(On Mach choose Document m Test Interface.)



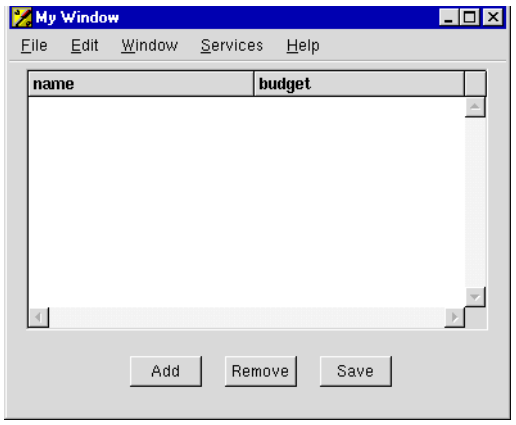
name	budget
20th Century Fox	$ 100,000,000.00
40 Acres and a Mule Filmwork	$ 81,000,000.00
AFI	$ 4,000,000.00
Alexandre Films	$ 26,000,000.00
Avenue Pictures	$ 9,000,000.00
British Lion Film Corp.	$ 2,000.00
Cargo Films	$ 1,000,000.00
Carolco Pictures	$ 6,000,000.00

Adding Action Methods

You can add basic behavior to your application, such as giving it the ability to add, delete, and save objects, without writing a line of code. This is possible because the EODisplayGroup and EOEditingContext objects in Interface Builder have predefined action methods that you can use to trigger operations in your application. An action method is a method that's invoked when the user clicks a button or another control object.

3 **Add action methods.**

Add three buttons to your window and label them "Add," "Remove," and "Save."



These buttons will be used to insert new studios, delete existing studios, and save changes.

Control-drag from the Add button to the Studio EODisplayGroup.

In the Inspector, select Outlets from the pop-up menu at the top of the left column.

Select **target** in the left column.

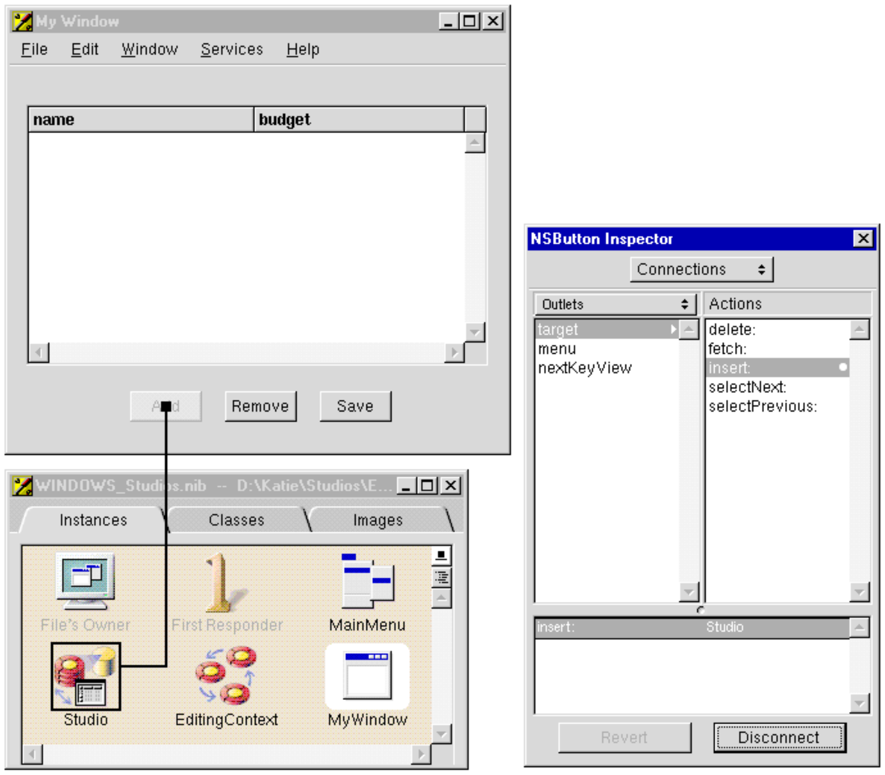
Double-click **insert:** in the right column.

Using the same process, connect the Remove button to the **delete:** method.

To connect the Save button, control-drag from the button to the EditingContext object in the nib file window.

In the Inspector, select **target** in the left column.

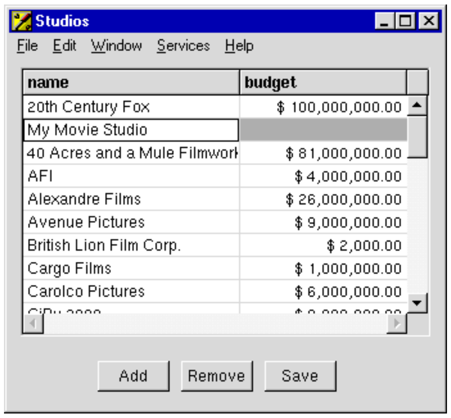
Double-click **saveChanges:** in the right column.



Now test-run your application in Interface Builder and try inserting and deleting some Studio objects. The changes you make aren't saved to the database until you click the Save button.

4 **Test your interface.**

Choose File m Test Interface.



name	budget
20th Century Fox	$ 100,000,000.00
My Movie Studio	
40 Acres and a Mule Filmwork	$ 81,000,000.00
AFI	$ 4,000,000.00
Alexandre Films	$ 26,000,000.00
Avenue Pictures	$ 9,000,000.00
British Lion Film Corp.	$ 2,000.00
Cargo Films	$ 1,000,000.00
Carolco Pictures	$ 6,000,000.00

What if It Doesn't Work?

What if you test-run the application at this point and it doesn't work?

- If no data appears in the table view, look in the Interface Builder Inspector to make sure that you have "Fetch on load" enabled for the Studio EODisplayGroup.
- If the buttons don't have the desired effect, check to see that they're connected to the appropriate action method in the appropriate object.
- If you get database errors when you try to add and delete studios or save changes, make sure that your model is properly specified. In particular, check that all of your entities have primary keys. Finally, choose Check Consistency from the Model menu in EOModeler to confirm that there are no problems in your model.

Optional Exercise

Enterprise Objects Framework provides additional action methods that you can use in connections: **fetch:** (EODisplayGroup) and **undo:**, **redo:**, **revert:**, and **refetch:** (EOEditingContext). Try adding controls (such as buttons or menu items) to the application and connecting them to some of these action methods.

Until now you have still not written a single line of code. However, because of the built-in features of Enterprise Objects Framework, all of the following have been provided for you:

- Automatic primary key generation when you insert a new object

  As described in the section "Assigning Primary Keys" on page 21, every row in a database is uniquely identified by its primary key value. When you create a new object in your application and save it to the database, you're adding a new row to a database table, and this row needs a primary key (that is, it needs to have a unique value for the primary key attribute you set in EOModeler). Enterprise Objects Framework handles generating this unique value for you.
- Formatting of money and dates
- Coordinating the user interface with your data

  Enterprise Objects Framework keeps all parts of an application in sync with the current view of the data. For example, if you have two windows in an application that are displaying the same data and you change the values in one window, the other will automatically update to reflect the changes.

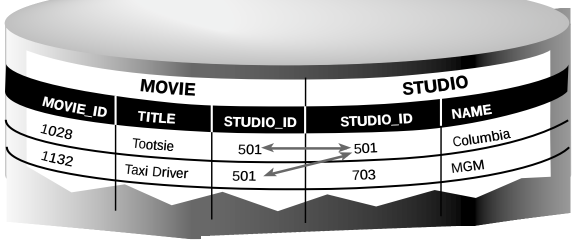
Adding Relationships

Creating an application that adds and modifies studios is just the first stage of the Studios application. Now you can enhance the application to display all of the movies owned by a selected studio. But in order to make this possible, you first need to go back to EOModeler and add relationships to the model's entities.

The Studio, Movie, and Talent entities are not especially interesting when considered separately. Their real significance only becomes apparent in their relationships to each other. Every Movie has one corresponding Studio. One Studio can have many Movies. A particular actor (Talent) can star in several movies.

Relational databases model not just individual entities, but entities' relationships to one another. For example, a Movie entity has a corresponding Studio entity. This is modeled in the database by both the Movie entity and the Studio entity having a **studioID** attribute. In Movie, **studioID** is a foreign key, while in Studio it's a primary key. A foreign key correlates with the primary key of another table in order to model a relationship a source table (Movie) has to a destination table (Studio). In the following diagram, notice that the value in the **STUDIO_ID** column for both movies is "501". This matches the value in the **STUDIO_ID** column of the Columbia Pictures movie studio. In other words, the movies "Tootsie" and "Taxi Driver" both belong to Columbia Pictures.

The value of the STUDIO_ID foreign key for the movies "Tootsie" and "Taxi Driver" matches the value of the STUDIO_ID primary key for Columbia Pictures.



This plays out in your running application as follows: Suppose you fetch a Movie object. Enterprise Objects Framework takes the value for the movie's **studioID** attribute and looks up the studio with the corresponding primary key.

For your application to take advantage of such database-defined relationships, you need to explicitly add a corresponding relationship to your model.

At this point you need to specify the following relationships:

From the Studio (source) entity:

- Form a *to-many* relationship to the Movie (destination) entity.
- The source attribute is **studioID**. The destination attribute is **studioID**.
- Name the relationship **movies**.

From the Movie (source) entity:

- Form a *to-one* relationship to the Studio (destination) entity.
- The source attribute is **studioID**. The destination attribute is **studioID**.
- Name the relationship **studio**.

1 **Create a relationship.**

Display the attributes view for the entity you want to use as the source of the relationship.

Choose Property m Add Relationship.

In the Relationship Inspector, select a destination entity.

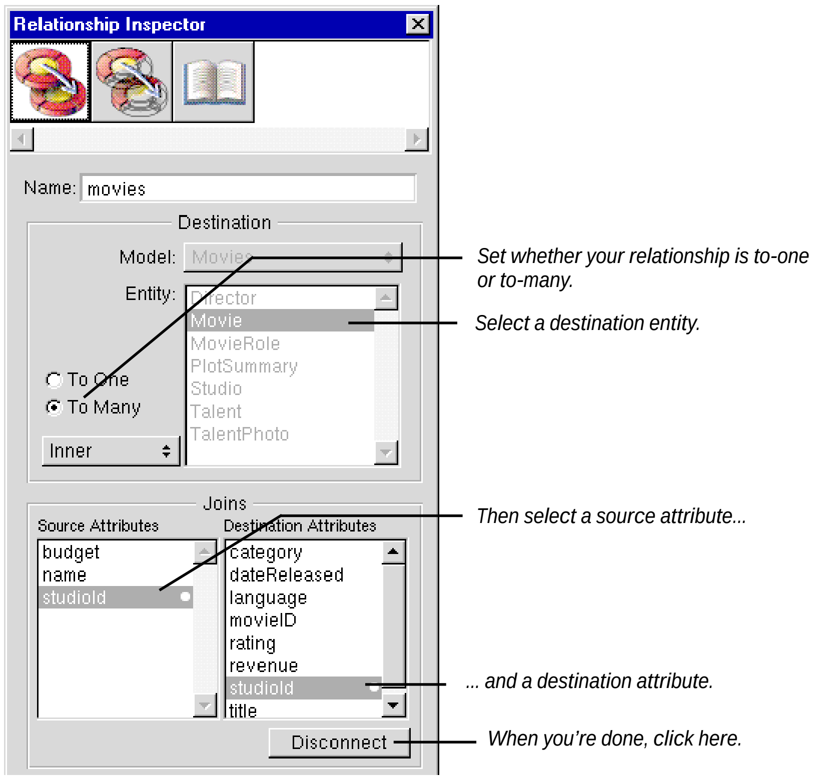
Select a source attribute.

Select a destination attribute.

Connect them.

2 **Name the relationship.**

Type the name in the Name field (where the text “Relationship” appears by default) and press Return.



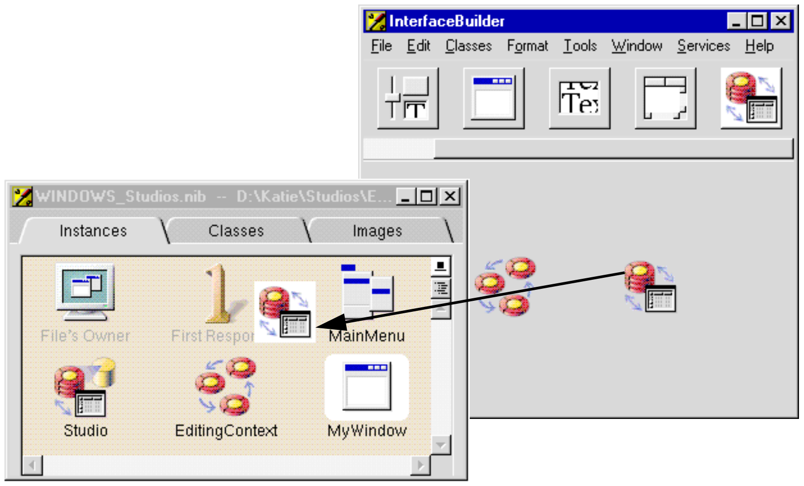
Adding Movies to the Application

The relationships you specified in EOModeler now come into play in your application. In EOModeler you added a to-many relationship from Studio to Movie, because a Studio can have many Movies. You can now use this relationship to display the movies for the selected studio.

In this type of configuration, called *master-detail*, the master table holds records for the source of the relationship, while the detail table contains records for the destination. In the Studios application, Studio is the master table and Movie is the detail table.

3 **Create a master-detail interface.**

From the EOPalette, drag an EODisplayGroup object into the nib file window.



The EODisplayGroup object you just dragged in from the EOPalette has no EODatabaseDataSource, unlike the Studio EODisplayGroup.

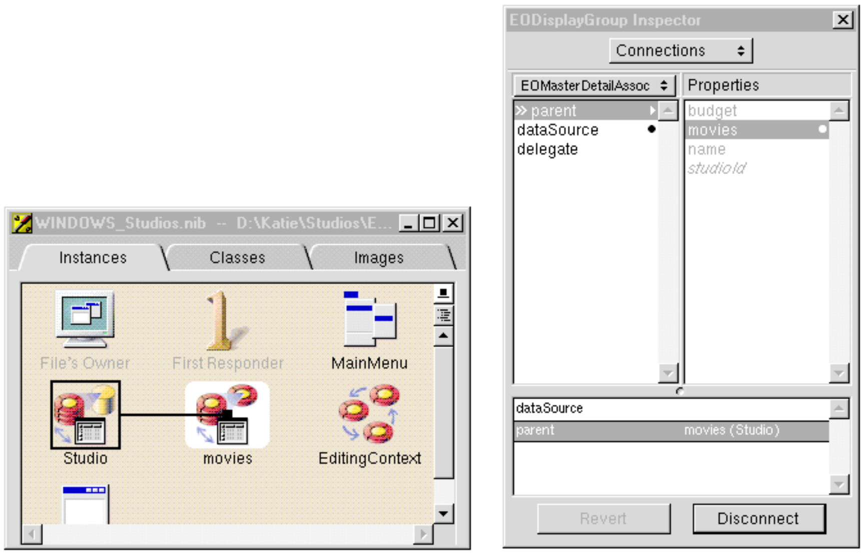
Control-drag from the detail (Movie) EODisplayGroup to the master (Studio) EODisplayGroup.

In the Inspector, choose EOMasterDetailAssoc from the pop-up list at the top of the left column.

Select **parent** in the left column.

Select **movies** in the right column. **movies** is the to-many relationship that Studio has to Movie.

Click Connect. The name of the EODisplayGroup changes to **movies**.

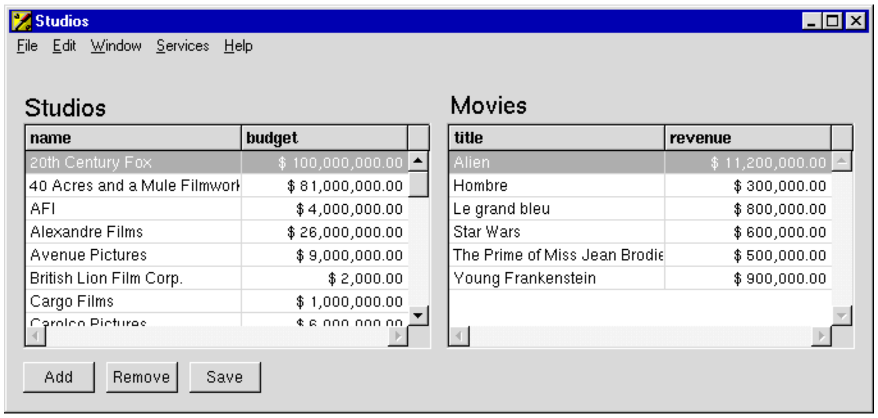


You can now use the movies EODisplayGroup to access all of the properties in the Movie object.

Drag a new table view into the window, and connect its columns to the **title** and **revenue** keys in Movie. You use the same process that was described in the section “Connecting the Studio EODisplayGroup to the Interface” on page 27.

Once you’ve connected the columns in the new table view to the movies EODisplayGroup, test the modified interface by choosing File m Test Interface.

Notice that labels have been added to the interface to identify the Studio and Movie tables.



name	budget
20th Century Fox	$ 100,000,000.00
40 Acres and a Mule Filmwork	$ 81,000,000.00
AFI	$ 4,000,000.00
Alexandre Films	$ 26,000,000.00
Avenue Pictures	$ 9,000,000.00
British Lion Film Corp.	$ 2,000.00
Cargo Films	$ 1,000,000.00
Carolco Pictures	$ 6,000,000.00

title	revenue
Alien	$ 11,200,000.00
Hombre	$ 300,000.00
Le grand bleu	$ 800,000.00
Star Wars	$ 600,000.00
The Prime of Miss Jean Brodie	$ 500,000.00
Young Frankenstein	$ 900,000.00

Note that when you select a studio in the Studios table view, the display changes in the Movies table view to show the selected studio’s movies.

Note: You can also create a master-detail interface by simply dragging a relationship from EOModeler into your window in Interface Builder. For a description of this, see the chapter “Creating an Enterprise Objects Framework Project.”

Transferring Movies Between Studios

One of the primary functions of the Studios application is to allow one studio to purchase movies from another. To make this possible, you’ll now add a pop-up list to the user interface.

The pop-up list displays a list of all of the studio titles. When you select a new studio in the pop-up list, you cause that studio to purchase the movie that’s selected in the table view.

1 **Add a pop-up list.**

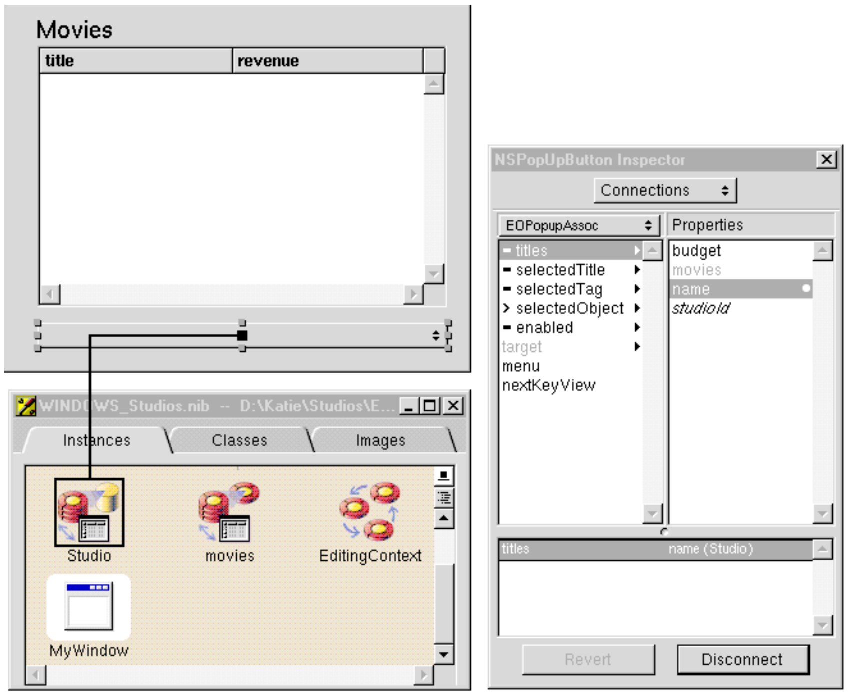
Drag a pop-up list into the window.

Control-drag from the pop-up list to the Studio EODisplayGroup.

In the Inspector, select EOPopupAssoc from the pop-up list at the top of the left column.

Select **titles** in the left column. The **titles** aspect is bound to the class key whose values you want to display in the pop-up.

Select **name** in the right column (since you want to display Studio names in the pop-up).



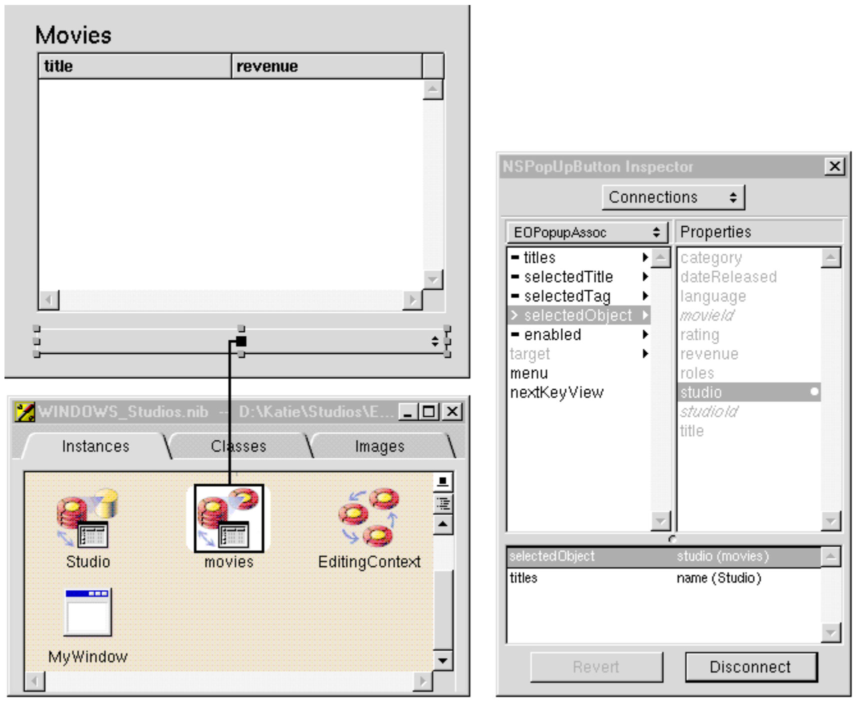
Now you have to add another binding to the EOPopupAssociation so that when you change the selected studio title, it sets the corresponding **studio** relationship property in the selected Movie object.

Control-drag from the pop-up list to the movies EODisplayGroup.

In the Inspector, select EOPopupAssoc from the pop-up list at the top of the left column.

Select **selectedObject** in the left column. The **selectedObject** aspect is bound to the relationship property (in this example, Movie's **studio** property) that corresponds to the object bound to the **titles** aspect (Studio).

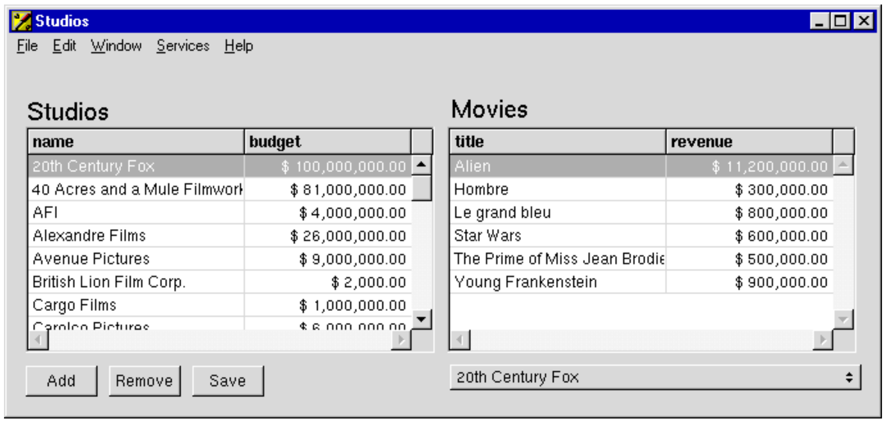
Select **studio** in the right column (since you want to change the Movie object's studio).



selectedObject	studio (movies)
titles	name (Studio)

2 **Test your interface and try out the new pop-up list.**

Choose File m Test Interface.



Studios

name	budget
20th Century Fox	$ 100,000,000.00
40 Acres and a Mule Filmwork	$ 81,000,000.00
AFI	$ 4,000,000.00
Alexandre Films	$ 26,000,000.00
Avenue Pictures	$ 9,000,000.00
British Lion Film Corp.	$ 2,000.00
Cargo Films	$ 1,000,000.00
Carolco Pictures	$ 6,000,000.00

Movies

title	revenue
Alien	$ 11,200,000.00
Hombre	$ 300,000.00
Le grand bleu	$ 800,000.00
Star Wars	$ 600,000.00
The Prime of Miss Jean Brodie	$ 500,000.00
Young Frankenstein	$ 900,000.00

You can now test the behavior of the pop-up list. For example, suppose you want to transfer the movie "Alien" from the 20th Century Fox studio to MGM. First select 20th Century Fox to display its movies. Then select "Alien" in the list of movies. Finally, use the pop-up list to change the selected studio from 20th Century Fox to MGM. This has the effect of removing "Alien" from 20th Century Fox's **movies** relationship array and adding it to the **movies** relationship array of MGM. It also sets the "Alien" Movie object's **studio** relationship property to point to the new studio, MGM. When you use the pop-up list to transfer a movie, you'll notice that the movie disappears from the original studio's movie list and reappears in the movie list of the new studio.

These changes aren't committed to the database until you click Save. At that time Enterprise Objects Framework translates the changes you made in the object graph into the appropriate database changes. For example, it sets the foreign key **studioID** in the transferred Movie object to have the same value as the **studioID** primary key of its new studio.

Note that Enterprise Objects Framework manages all of this for you without requiring you to write any code.

Putting the Finishing Touches on Your Model

You are almost ready to add custom behavior to your enterprise objects. But first you need to put a few finishing touches on your model.

In "Adding Relationships" on page 35, you added relationships between the Studio and Movie entities. Now you need to add a few additional relationships to your model:

From the Movie (source) entity:

- Form a *to-many* relationship to the MovieRole (destination) entity.
- The source attribute is **movieID**. The destination attribute is **movieID**.
- Name the relationship **roles**.

- Form a *to-one* relationship to the PlotSummary (destination) entity.
- The source attribute is **movieID**. The destination attribute is **movieID**.
- Name the relationship **plotSummary**.

From the Talent (source) entity:

- Form a *to-many* relationship to the MovieRole (destination) entity.
- The source attribute is **talentID**. The destination attribute is **talentID**.
- Name the relationship **roles**.

- Form a *to-one* relationship to the TalentPhoto (destination) entity.
- The source attribute is **talentID**. The destination attribute is **talentID**.
- Name the relationship **photo**.

From the MovieRole (source) entity:

- Form a *to-one* relationship to the Movie (destination) entity.
- The source attribute is **movieID**. The destination attribute is **movieID**.
- Name the relationship **movie**.

- Form a *to-one* relationship to the Talent (destination) entity.
- The source attribute is **talentID**. The destination attribute is **talentID**.
- Name the relationship **talent**.

Removing Primary and Foreign Keys as Class Properties

By default, EOModeler makes all of an entity's attributes class properties (except for non-database attributes that you add to the entity). When an attribute is a class property, it means that the property will be included in your class definition and that it can be fetched from the database. To put it another way, only attributes that are marked as class properties become part of your enterprise objects.

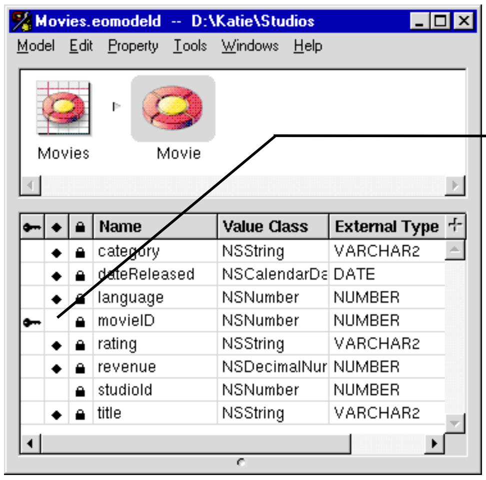
You should only mark as class properties those attributes whose values are meaningful in the objects that are created when you fetch from the database. Attributes that are essentially database artifacts, such as primary and foreign keys, shouldn't be marked as class properties unless the key has meaning to the user and must be displayed in the user interface.

Eliminating primary and foreign keys as class properties has no adverse effect on how Enterprise Objects Framework manages enterprise objects in your application.

3 **Remove primary and foreign keys as class properties.**

In the Model Editor, select the entity you want to modify.

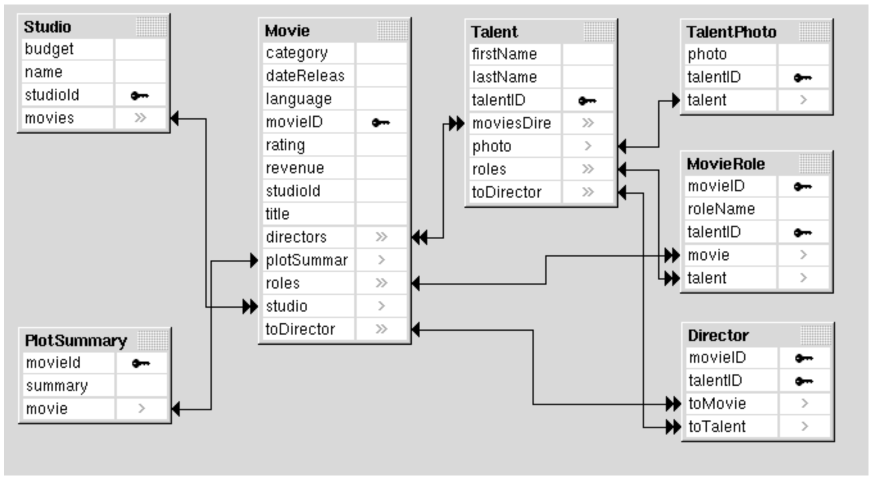
Click in the Class Property column to remove the ◆ symbol.


Movies.eomodeld -- D:\Katie\Studios
Model Edit Property Tools Windows Help
Movies
Movie

🔑	◆	🔒	Name	Value Class	External Type
	◆	🔒	category	NSString	VARCHAR2
	◆	🔒	dateReleased	NSCalendarDa	DATE
	◆	🔒	language	NSNumber	NUMBER
🔑		🔒	movieID	NSNumber	NUMBER
	◆	🔒	rating	NSString	VARCHAR2
	◆	🔒	revenue	NSDecimalNur	NUMBER
		🔒	studioId	NSNumber	NUMBER
	◆	🔒	title	NSString	VARCHAR2

Click in an attribute's Class Property column to remove it as a class property.

At this point your model is complete. Looking at your model using the Diagram View (available through ModelerBundle in the on-line examples) gives you an overview of the entities in the model and their relationships to other entities.



Adding Behavior to Your Enterprise Objects

As the preceding sections illustrate, you can go quite far in an Enterprise Objects application without writing any code.

However, the real power of an Enterprise Objects Framework application lies in the enterprise objects you create. The behavior (business logic) you add to your objects is what brings your stored data to life.

Specifying Custom Enterprise Object Classes for Studio and Talent

Unless you specify otherwise, EOModeler maps entities to the EOGenericRecord class, which can be thought of as the default enterprise object class.

However, when you want to add custom behavior to a class (for example, to assign default values when you create new objects or to perform validation), you need to implement a custom enterprise object class. This class includes the default behavior provided in EOGenericRecord as well as the custom behavior you implement.

When Do You Use a Custom Enterprise Object Class?

Enterprise Objects Framework provides a "default" enterprise object class, EOGenericRecord. An EOGenericRecord can take on values for any properties defined in your application's model, but it implements no custom behavior. EOGenericRecord objects can hold simple values as well as refer to other enterprise objects through relationships defined in the model.

The criterion for deciding whether to make your enterprise objects custom classes or to simply use the EOGenericRecord class is *behavior.* One of the main reasons to use the Enterprise Objects Framework is to associate behavior with your persistent data. Behavior is implemented as methods that "do something" (as opposed to merely setting or returning the value for a property). Since the Framework itself handles most of the behavior related to persistent storage, you can focus on the behavior specific to your application.

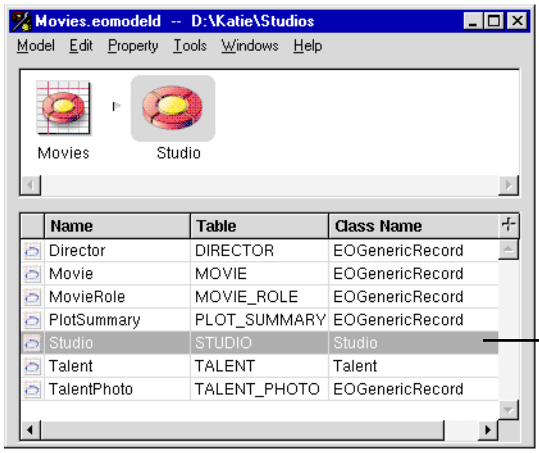
Because the Studio and Talent classes need to have specialized behavior (for example, to perform validation when you attempt to save changes to the database), they need to be custom classes.

1 **Specify custom enterprise object classes.**

In the Model Editor, select the entity for which you want to specify an enterprise object class.

Specify an enterprise object class for the entity by typing the class name in the Class Name field.

For the Studios application, you need to create custom classes for both Studio and Talent. These classes should be named **Studio** and **Talent**, respectively.


Movies.eomodeld -- D:\Katie\Studios
Model Edit Property Tools Windows Help
Movies
Studio

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

Replace the text "EOGenericRecord" with the name want to use for the class.

You need to perform this operation for the Studio and Talent entities. Movie doesn't need to be a custom class since it doesn't have any specialized behavior. By convention, class names are capitalized and based on the name of corresponding entity. Consequently, you should name the classes **Studio** and **Talent**.

Once you specify a custom class for an entity in EOModeler, you can generate template source code for that entity.

Generating Template Source Code

To begin creating your custom classes, generate template source code for the Studio and Talent entities. You use this template source code as a basis for adding your own methods to your enterprise objects.

Generating template files produces:

- A header (**.h**) file that declares instance variables for all of the class properties, and optional accessor methods for those instance variables. Accessor methods set and return the values of the object's instance variables.
- An implementation (**.m**) file that provides basic implementations for the accessor methods.

Note: To generate template source code for an entity, you must have provided a name for it (that is, you must have replaced the text "EOGenericRecord") in the Class Name field.

1 **Generate template source code.**

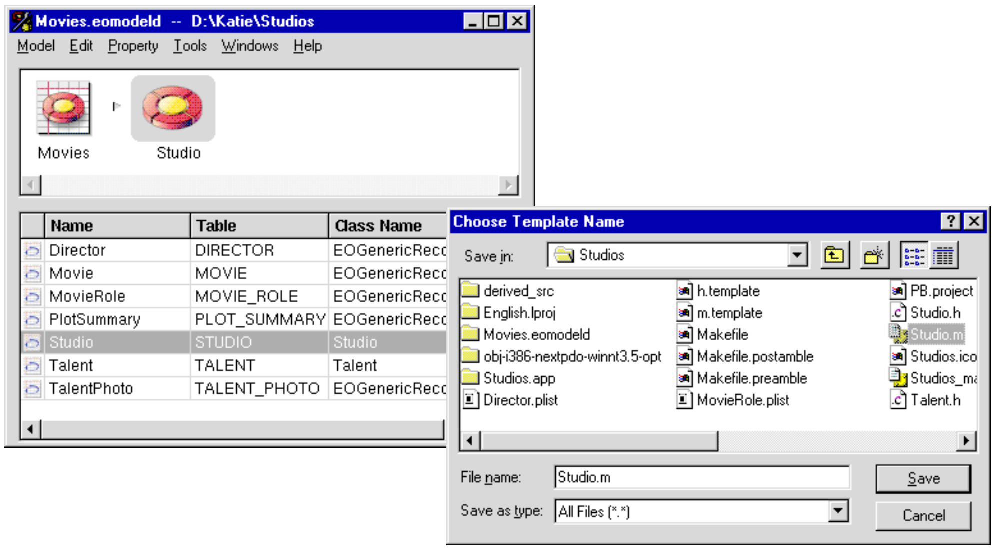
In the Model Editor, select the entity for which you want to generate template source code.

Choose Property ➞ Create Template from the Property menu. 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, and **<*Class*>.m** as the default file name. Click Save.

A panel appears, asking if you want to insert the files in your project. Click OK.

You need to generate template source code for both the Studio and Talent entities.


Movies.eomodeld -- D:\Katie\Studios
Model Edit Property Tools Windows Help
Movies
Studio
Choose Template Name
Save in:
Save
Cancel

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

Save in: Studios

derived_src	h.template	PB.project
English.lproj	m.template	Studio.h
Movies.eomodeld	Makefile	Studio.m
obj-i386-nextpdo-winnt3.5-opt	Makefile.postamble	Studios.ico
Studios.app	Makefile.preamble	Studios_ma
Director.plist	MovieRole.plist	Talent.h

File name: Studio.m

Save as type: All Files (*.*)

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

Studio.h

```
@interface Studio : NSObject
{
    NSDecimalNumber *budget;
    NSString *name;
    NSMutableArray *movies;
}

- (void)setBudget:(NSDecimalNumber *)value;
- (NSDecimalNumber *)budget;

- (void)setName:(NSString *)value;
- (NSString *)name;

- (NSArray *)movies;
- (void)addToMovies:(Movie *)object;
- (void)removeFromMovies:(Movie *)object;

@end
```

Implementing Custom Behavior for Your Classes

The user interface you designed in Interface Builder already allows you to insert and delete Studio objects. However, it doesn't do any additional processing when these operations take place. For example, what if you want to assign default values to newly created objects? And how can you prevent users from inserting objects that contain invalid data? You can add methods to your enterprise objects to handle such issues.

Adding Behavior to Enterprise Objects

Some of the more common ways to add behavior to your enterprise object classes are:

- Performing computations based on the values of class properties. For example, from an Employee's salary property, you might calculate a bonus.
- Managing the creation and insertion of objects (for example, assigning default values to newly created objects, creating related objects as the by-product of inserting a new object, appropriately setting relationships for new objects, and so on)
- Performing validation when a particular operation (such as save or delete) takes place
- Adding sophisticated business logic

For a more complete discussion of this subject, see the chapter "Designing Enterprise Objects."

Managing Relationships

In the section "Transferring Movies Between Studios" on page 38, you added a pop-up list to the user interface to transfer movies between studios. However, there is still work to be done. When a movie is sold to a new studio, you need to add the amount of the movie's revenue to the old studio's budget (to show the studio's profit from the sale). Likewise, you need to subtract the amount of the movie's revenue from the new studio's budget (to reflect the expense of purchasing the movie).

When you transfer movies between studios, you're actually manipulating the **movies** relationship property in each of the Studio objects, deleting the Movie object from the **movies** array of the old studio, and adding the Movie object to the **movies** array of the new studio. Enterprise Objects Framework automatically invokes the method **addObject:ToPropertyWithKey:** when you add an object to an array that represents a relationship property, and invokes **removeObject:fromPropertyWithKey:** when you delete an object from the array. These methods are part of the EOKeyRelationshipManipulation protocol; for more information see the NSObject Additions class specification in the EOControl framework.

When passed a key (such as **movies**), the default implementations of these methods look for a method that has the name **addTo***Key***:** (when an object is being added) and **removeFrom***Key***:** (when an object is being removed). Skeletal versions of these methods are provided in your template source code.

To intervene and perform your own processing when objects are added to and removed from the **movies** relationship array, you add code to the methods **addToMovies:** and **removeFromMovies:** in the Studio class:

```
- (void)addToMovies:(id)movie
{
    NSDecimalNumber *newBudget;
    [self willChange];
    newBudget = [[self budget] decimalNumberBySubtracting:
        [movie valueForKey:@"revenue"]]
    [self setBudget:newBudget];
    [movies addObject:movie];
}

- (void)removeFromMovies:(id) movie
{
    NSDecimalNumber *newBudget;
    [self willChange];
    newBudget = [[self budget] decimalNumberByAdding:
        [movie valueForKey:@"revenue"]]
    [self setBudget:newBudget];
    [movies removeObject:movie];
}
```

Writing Derived Methods

One kind of behavior you might want to add to your enterprise object class is the ability to perform computations based on the values of class properties. For example, studios have movies, and the total revenue of the movies constitutes the studio's portfolio value. To calculate a studio's portfolio value, you could have a method in **Studio.m** like the following:

```
- (NSDecimalNumber *)portfolioValue
{
    NSDecimal result = [[NSDecimalNumber zero] decimalValue];
    int i = [movies count];
    while( i-- ) {
        NSDecimalNumber *value = [[movies objectAtIndex:i]
             valueForKey:@"revenue"];
        if(value) {
             NSDecimal total = [value decimalValue];
             NSDecimalAdd(&result, &result, &total, NSRoundBankers);
        }
    }
    return [NSDecimalNumber decimalNumberWithDecimal:result];
}
```

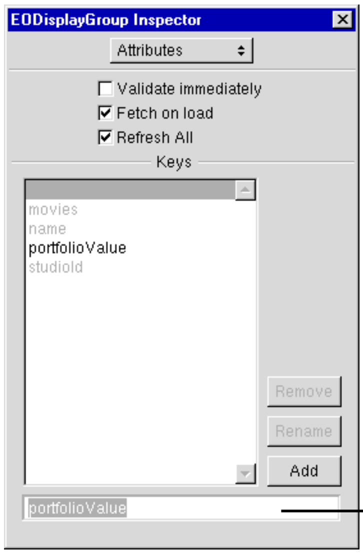
You can display the results of this method in the user interface by forming an association between a control and the method. That way, whenever a new studio is selected or when a selected studio's movie revenues change, its portfolio value is dynamically recalculated and displayed.

2 **Associate a method with a user interface control.**

Display the Attributes view of the Inspector for the Studio EODisplayGroup.

Add the name of the method (**portfolioValue**) you want to use in an association.

Click Add.



Type the method name here.

Once you've added the method as a class key, you can use it in associations.

Drag a text field into the window, and add a Currency formatter to it.

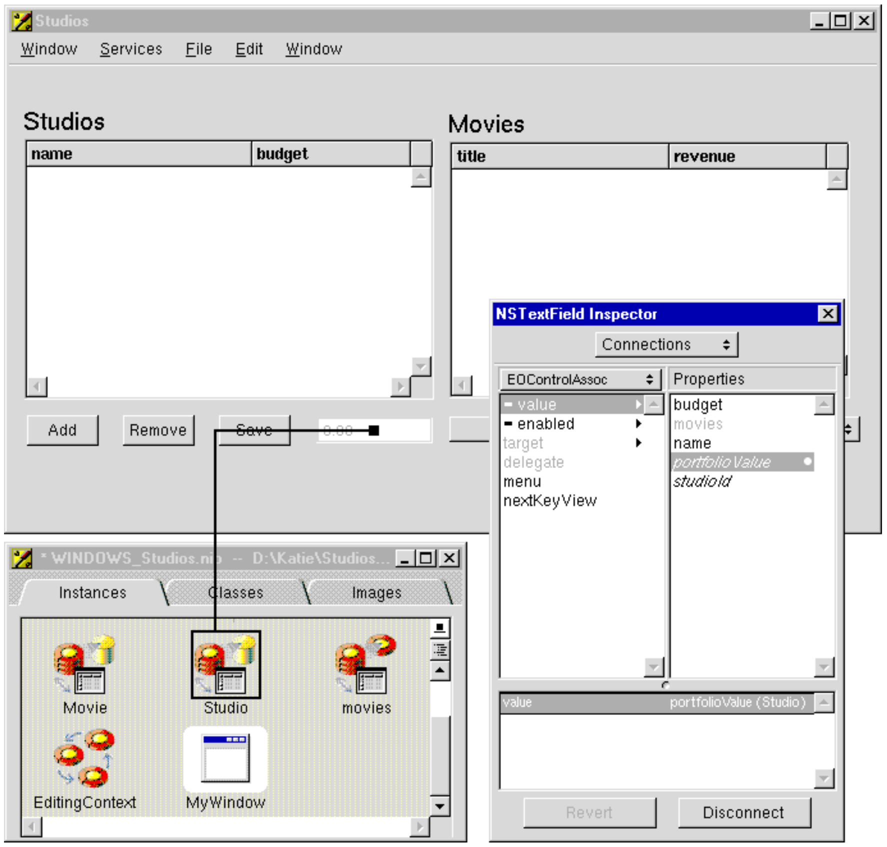
Control-drag from the text field to the Studio EODisplayGroup.

In the Connections Inspector, choose EOControlAssoc from the pop-up list at the top of the left column.

Select **value** in the left column.

In the right column select the method (**portfolioValue**) you want to associate with the control.

Double-click **portfolioValue** to connect.

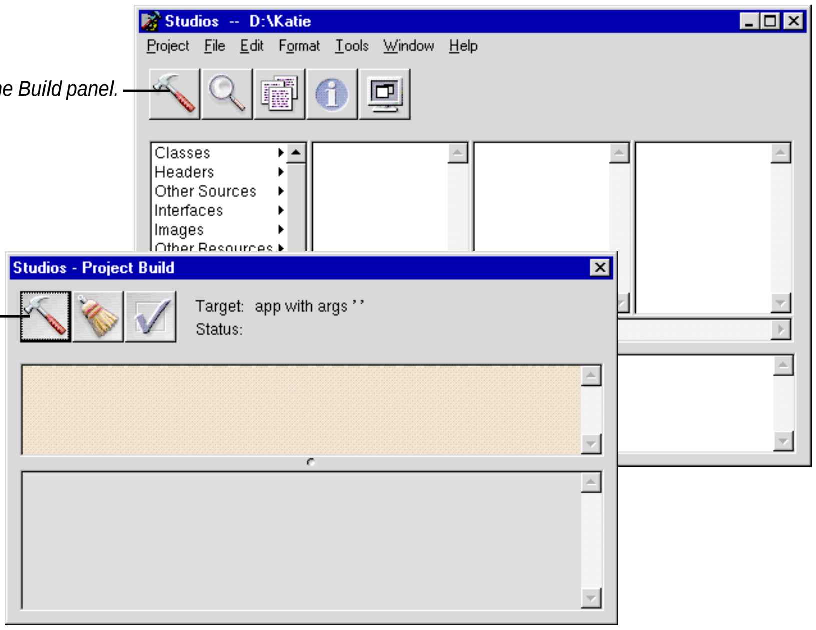


To see the effects of the change, you must compile and run the application.

3 **Use Project Builder to build the application.**

Click here to display the Build panel.

Click here to build your application.



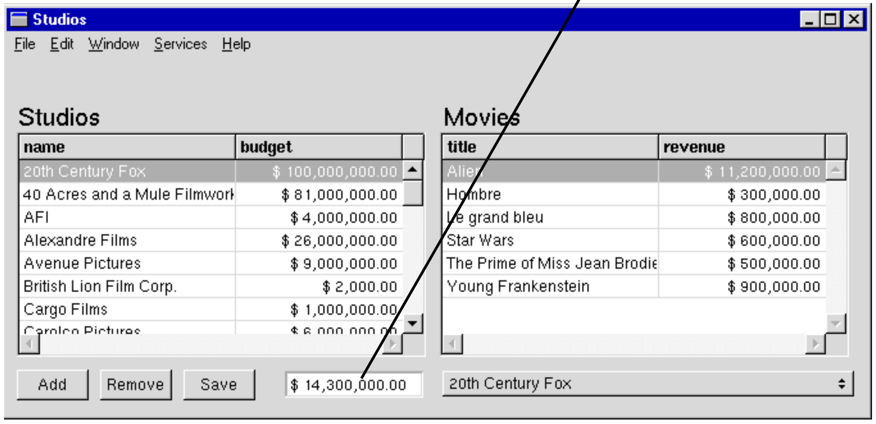
Target: app with args ' '
Status:

You can now run the application to see the effects of the portfolioValue method.

4 **Run the application.**

Choose Tools ▸ Launcher ▸ Run, or else open the **Studios.app** directory and double-click **Studios.exe**.

The selected studio's portfolio value is displayed here. It's dynamically recalculated whenever you select a new studio or when the selected studio's portfolio value changes.



Studios

name	budget
20th Century Fox	$ 100,000,000.00
40 Acres and a Mule Filmwork	$ 81,000,000.00
AFI	$ 4,000,000.00
Alexandre Films	$ 26,000,000.00
Avenue Pictures	$ 9,000,000.00
British Lion Film Corp.	$ 2,000,000.00
Cargo Films	$ 1,000,000.00
Carolco Pictures	$ 6,000,000.00

$ 14,300,000.00

Movies

title	revenue
Alien	$ 11,200,000.00
Hombre	$ 300,000.00
Le grand bleu	$ 800,000.00
Star Wars	$ 600,000.00
The Prime of Miss Jean Brodie	$ 500,000.00
Young Frankenstein	$ 900,000.00

20th Century Fox

Performing Validation

Another element you'll likely want to add to your enterprise object classes is validation. For example, suppose that when a studio buys a new movie, you want to check to make sure that acquiring the movie won't cause the studio to exceed its budget. You could implement a method in the Studio class like the following:

```
- (NSException *)validateForSave
{
    if ([[self budget] doubleValue] < 0)
       return [NSException validationExceptionWithFormat:
             @"You're exceeding your budget!"];
    return [super validateForSave];
}
```

Now when a studio buys more movies than it can afford, a panel displaying the message "You're exceeding your budget!" appears when the user attempts to save the changes to the database.

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

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

For more discussion of this topic, see the chapter "Designing Enterprise Objects."

Providing Default Values for Newly Inserted Objects

When new objects are created in your application and inserted into the database, it's common to assign default values to some of their properties. For example, you might decide to assign newly created Studio objects a default budget (the budget is the amount a studio is allowed to spend on new movies).

To assign default values to newly created enterprise objects, use the method **awakeFromInsertionInEditingContext:**. This method is automatically invoked right after your enterprise object class creates a new object and inserts it into an EOEditingContext.

The following implementation of **awakeFromInsertionInEditingContext:** in the Studio class sets the default value of the **budget** property to be one million dollars:

```
- (void)awakeFromInsertionInEditingContext:(EOEditingContext *)ctx
{
     if (!budget)
        budget = [NSDecimalNumber
               decimalNumberWithString:@"1000000.00"];
    [super awakeFromInsertionInEditingContext:ctx];
}
```

When a user clicks the Add Studio button in the Studios application, a new record is inserted, with “$1,000,000.00” already displayed as a value in the **budget** column. Notice that because you put a formatter on this column, the value is formatted correctly after you insert the object.

Adding Business Logic

In addition to such operations as assigning default values to new objects and performing validation, enterprise objects can also implement more sophisticated business logic. For example, suppose you want to give studios the ability to buy all of the movies that star a specified actor. You can implement a method such as the following in **Studio.m**:

```
- (void)buyAllMoviesStarring:(Talent *)talent
{
    NSArray *actorsMovies = [talent moviesStarredIn];
    unsigned i = [actorsMovies count];
    while (i--) {
        id movie = [actorsMovies objectAtIndex:i];
        if (![movies containsObject:movie]) {
             [self addToMovies:movie];
       }
    }
}
```

This method invokes the **moviesStarredIn** method, which is implemented in **Talent.m**:

```
- (NSArray *)moviesStarredIn
{
    unsigned i = [roles count];
    NSMutableArray *result = [NSMutableArray arrayWithCapacity:i];
    while (i--) {
        id movie = [[roles objectAtIndex:i] valueForKey:@"movie"];
        if (![result containsObject:movie]) {
             [result addObject:movie];
       }
    }
    return result;
}
```

You can associate the **buyAllMoviesStarring:** method with a user interface control. But first you need to add to your user interface a table view that lists all actors (talent).

5 **Add a new table view to your user interface.**

Drag the Talent entity from your model into the nib file window in Interface Builder.

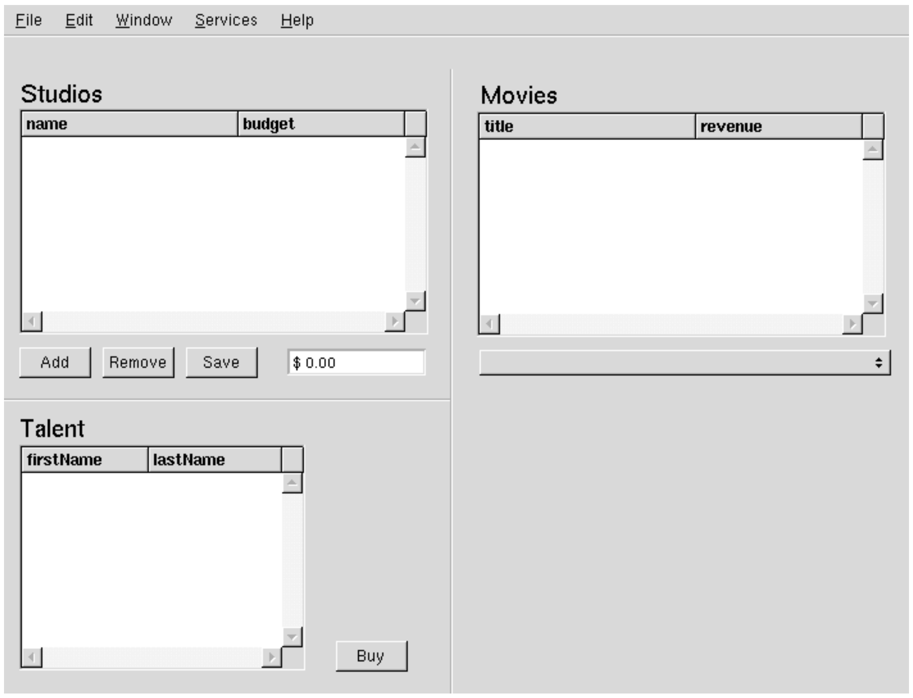
Add a table view to your window.

Control-drag from each table view column to the Talent EODisplayGroup.

Using the **value** aspect of the EOColumnAssoc, connect the table view columns to the **firstName** and **lastName** class keys, respectively.

6 **Add a button to the window.**

Drag a button into the window next to the Talent table view and label it "Buy".



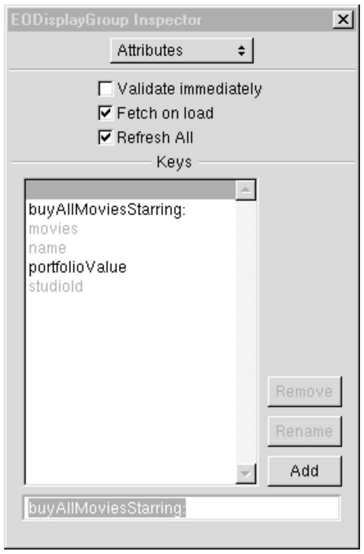
Now that you've added the table view, connected it to the **firstName** and **lastName** properties of the Talent EODisplayGroup, and added a Buy button to the window, you're ready to use an EOActionAssociation to connect the button to the **buyAllMoviesStarring:** method.

7 **Associate a method with a user interface control.**

Display the Attributes view of the Inspector for the Studio EODisplayGroup.

Add the name of the method (**buyAllMoviesStarring:**) you want to use in an association.

Click Add.


EODisplayGroup Inspector

Attributes

[ ] Validate immediately
[✓] Fetch on load
[✓] Refresh All

Keys

buyAllMoviesStarring:
movies
name
portfolioValue
studioId

buyAllMoviesStarring:

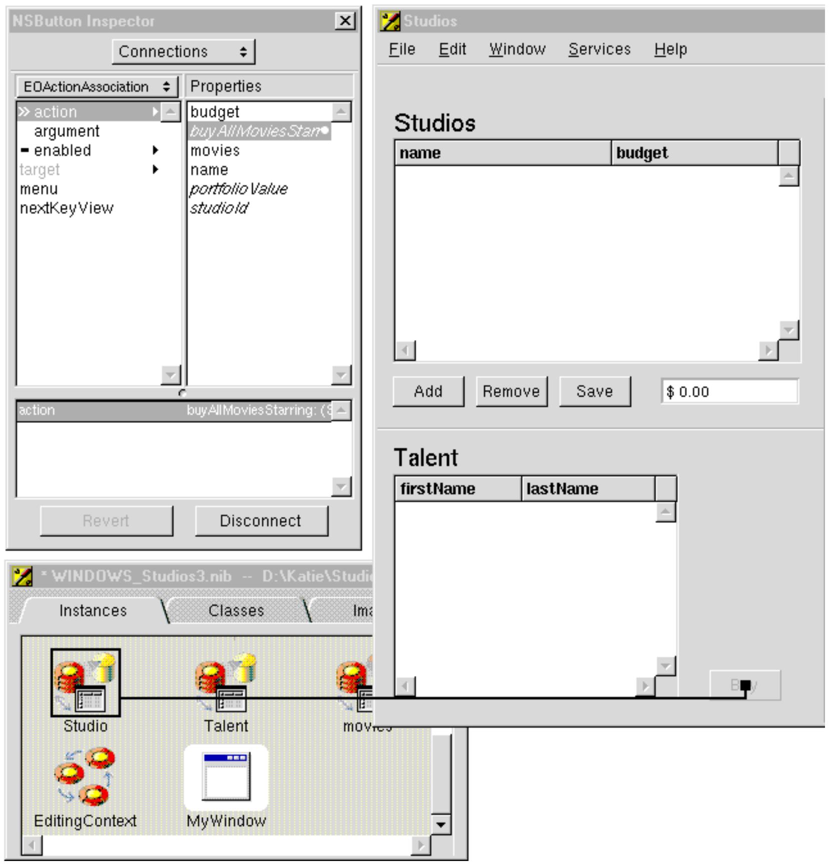
You can now use the **buyAllMoviesStarring:** method in associations.

Control-drag from the Buy button to the Studio EODisplayGroup.

In the Connections Inspector, choose EOActionAssociation from the pop-up list at the top of the left column.

Select **action** in the left column, and the method you want to connect to (**buyAllMoviesStarring:**) in the right column.

Click Connect.

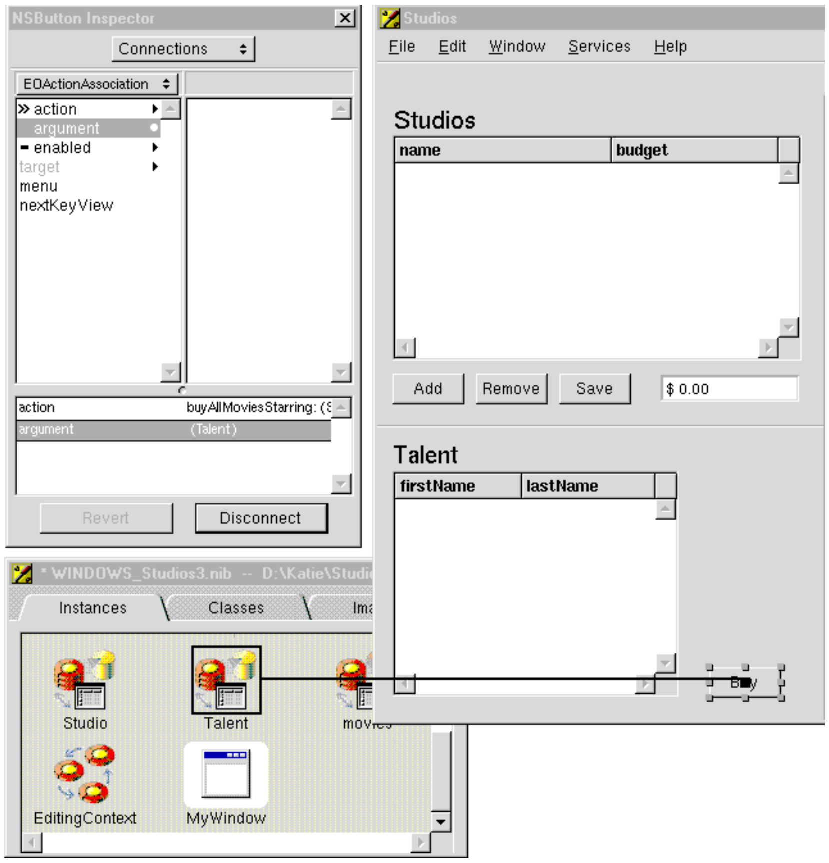


Because the **buyAllMoviesStarring:** method takes a Talent object as an argument, you need to make a connection from the Buy button to the Talent EODisplayGroup.

Control-drag from the Buy button to the Talent EODisplayGroup.

In the Inspector, select **argument** in the left column. The **argument** aspect takes the destination of the connection (Talent) as an argument, which will be supplied to the **buyAllMoviesStarring:** method.

Click Connect.



Once you finish connecting the button, you can use it to purchase all of the movies starring the selected actor for the selected studio.

Optional Exercise

You can display different types of data in your user interface. For example, you can add an image view to display the photo of a selected actor, and a scroll view to display the plot summary for the selected movie.

To use image data in your application, you must first have set up an NSImage custom value for TalentPhoto's photo attribute in your model. In the Attribute Inspector, use the pop-up list to set photo's data type to Custom. In the Class text field, type NSImage. In the Factory Method text field, type imageWithData:.In the Conversion Method text field, type TIFFRepresentation. Finally, use the Init Argument pop-up list to specify the type NSData.

To add an image view, drag an EODisplayGroup from the EOPalette into the nib file window. Control-drag to form an EOMasterDetailAssociation between it and the Talent EODisplayGroup; select parent in the left column and photo in the right. Then drag an image view object from the palette into the window. Control-drag to connect it to the new photo EODisplayGroup using an EOControlAssociation; select value in the left column and photo in the right.

To add a text view, drag an EODisplayGroup from EOPalette into the nib file window. Control-drag to form an EOMasterDetailAssociation between it and the movies EODisplayGroup; select parent in the left column and plotSummary in the right. Drag a scroll view into the window.

To connect the text view portion of the scroll view to the EODisplayGroup, you have to use the outline view of the nib file window. To navigate to the scroll view that contains the text view in the nib file window, place the cursor in the text view and press Command-e. This displays the outline view of the nib file window with the scroll view selected. In the nib file window, open the scroll view and select the text view it contains. Control-drag within the outline view of the nib file window to connect the text view to the new plotSummary EODisplayGroup using an EOControlAssociation; select value in the left column and summary in the right.

Running the Studios Application

Congratulations! You've just finished creating your first Enterprise Objects Framework application. Build your application using Project Builder, open the Studios.app directory in your project directory, start it by double-clicking Studios.exe, and try it out.

Although it's a simple example, the Studios application introduced you to many of the concepts, tools, and skills you'll need to create Enterprise Objects Framework applications. You've learned about:

- Creating a model using EOModeler
- Generating template source code for enterprise object classes
- Composing a graphical user interface for an Enterprise Objects Framework with Interface Builder
- Adding formatting to your user interface
- Creating a master-detail interface
- Using EODisplayGroup and EOEditingContext action methods

- Implementing custom behavior for enterprise object classes

A Quick Guide to Enterprise Objects Framework and Relational Database Terminology

Several of the terms listed here apply to relational databases and entity-relationship modeling. For a more complete discussion of this subject, see the appendix "Entity-Relationship Modeling."

attribute

In Entity-Relationship modeling, an identifiable characteristic of an entity. For example, **lastName** can be an attribute of an **Employee** entity. An attribute typically corresponds to a column in a database table. See *flattened attribute*, *entity*, and *relationship*.

class property

An instance variable in an enterprise object that meets two criteria: it's based on an attribute in your model, and it can be fetched from the database. "Class property" can either refer to an attribute or a relationship.

column

In a relational database, the dimension of a table that holds values for a particular attribute. For example, a table that contains employee records might have a column titled "LAST_NAME" that contains the values for each employee's last name. See *attribute*.

compound primary key

In a database table, the group of columns whose values, taken in combination, are guaranteed to uniquely identify each row. See *primary key*.

data dictionary

In relational databases, the system tables that describe the organization of data in a particular database.

database server

A data storage and retrieval system. Database servers typically run on a dedicated computer and are accessed by client applications over a network.

enterprise object

An Objective-C object that conforms to the key-value coding protocol, whose properties (data) can map to stored data. An enterprise object brings together stored data with the methods for operating on that data. See *key-value coding* and *property*.

entity

In Entity-Relationship modeling, a distinguishable object about which data is kept. For example, you can have an Employee entity with attributes such as lastName, firstName, address, and so on. An entity typically corresponds to a table in a relational database; an entity's attributes in turn correspond to a table's columns. See *attribute* and *table*.

Entity-Relationship modeling

A discipline for examining and representing the components and interrelationships in a database system. Also known as E-R modeling, this discipline factors a database system into entities, attributes, and relationships.

fetch

In Enterprise Objects Framework applications, to retrieve data from the database server into the client application, usually into enterprise objects.

flattened attribute

A special kind of attribute that you add from one entity to another by traversing a relationship. For example, employees work for departments; you can add an attribute (such as departmentName) from the Department entity to the Employee entity as a flattened attribute. A flattened attribute is normally implemented by joining the tables corresponding to the source and destination entities whenever the attribute's data is fetched. See *relationship* and *attribute*.

foreign key

An attribute in an entity that gives it access to rows in another entity. This attribute must be the primary key of the related entity. For example, an Employee entity can contain the foreign key deptID, which matches the primary key in the entity Department. You can then use deptID as the source attribute in Employee and as the destination attribute in Department to form a relationship between the entities. See *key*, *primary key*, and *relationship*.

generic record

An instance of the EOGenericRecord default enterprise object class. A generic record has properties that map to stored data, but unlike a custom enterprise object, it adds no behavior to that data. Like custom enterprise objects, generic records conform to the key-value coding protocol; see *key-value coding*.

join

An operation that provides access to data from two tables at the same time, based on the values contained in related columns.

key-value coding

The mechanism that allows the properties in enterprise objects to be accessed by name (that is, as key-value pairs) by other parts of the Framework.

many-to-many relationship

A relationship in which each record in the source entity may correspond to more than one record in the destination entity, and each record in the destination may correspond to more than one record in the source. For example, an employee can work on many projects, and a project can be staffed by many employees. See *relationship*.

model

An EOModel object that defines, in Entity-Relationship terms, the mapping between enterprise object classes and the database schema. This definition is typically stored in a file created with the EOModeler application. A model also includes the information needed to connect to a particular database server; see *connection dictionary*.

record

The set of values that describes a single instance of an entity; in a relational database, a record is equivalent to a row.

relational database

A database designed according to the relational model, which uses the discipline of Entity-Relationship modeling and the data design standards called normal forms.

relationship

A link between two entities that's based on attributes of the entities. For example, the Department and Employee entities can have a relationship based on the deptID attribute as a foreign key in Employee, and as the primary key in Department (note that though the join attribute deptID is the same for the source and destination entities in this example, it doesn't have to be). This relationship would make it possible to find the employees for a given department. See *to-one, to-many, many-to-many, primary key*, and *foreign key*.

row

In a relational database, the dimension of a table that groups attributes into records.

table

A two-dimensional set of values corresponding to an entity. The columns of a table represent characteristics of the entity and the rows represent instances of the entity.

to-many relationship

A relationship in which each source record has zero to many corresponding destination records. For example, a department has many employees

to-one relationship

A relationship in which each source record has exactly one corresponding destination record. For example, each employee has one job title.

.