Chapter 4

Creating an Enterprise Objects Framework Project

Organizing an Enterprise Objects Framework application, as with all applications in OpenStep, starts in Project Builder.

After you use Project Builder to prepare your project for the needs of an Enterprise Objects Framework application, you design the application's user interface with Interface Builder and write its code.

This chapter describes the things you do in Project Builder and Interface Builder to create an Enterprise Objects Framework application. It's assumed that you have some familiarity with these applications. For more information about them, see *OPENSTEP Development: Tools & Techniques* (on Windows NT, this information is in Windows Help for Interface Builder and Project Builder).

The interface layer of the Enterprise Objects Framework allows you to create user interfaces for any enterprise object class. The examples in this chapter are based on creating user interfaces for enterprise object classes specified in EOModeler. Using EOModeler and Interface Builder together automates the development process significantly. For more information on creating a model, see the chapter "Using EOModeler."

Creating a Project

To create your project, start Project Builder and choose Project ▸ 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.

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.

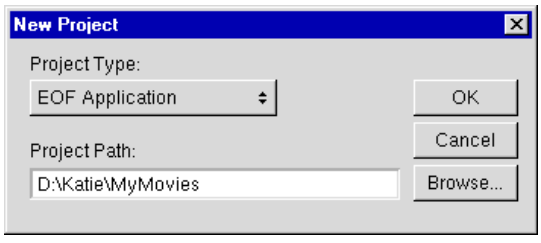

Figure 32. *Creating a New Project*

Project Builder creates a project directory named after the project—in this case MyMovies—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 Interface

To begin creating a user interface for your application, open your project and select Interfaces in the project window. Double-click on your project's nib file to open it in Interface Builder. On Windows NT this is WINDOWS_*MyApp*.nib; on Mach it is NEXTSTEP_*MyApp*.nib. From there you can start creating the user interface for your application.

The interface objects that are provided by the Enterprise Objects Framework for use in Interface Builder are manipulated in the same manner as standard Application Kit objects such as Buttons and Sliders: You drag an object into a window and drop it on or connect it to some other object.

Loading EOPalette

The palette provided for use in the Enterprise Objects Framework is the EOPalette. To load the EOPalette, choose Tools m Palettes m Open.

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

The EOPalette includes two objects: EODisplayGroup and EOEditingContext.

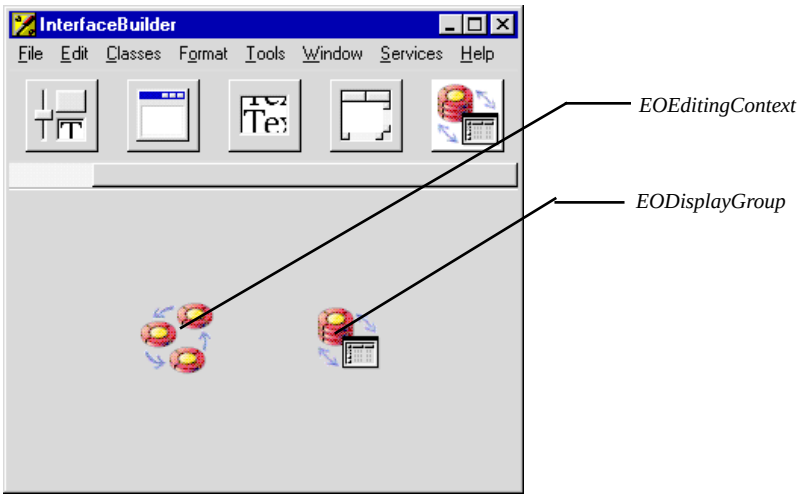

Figure 33. *The EOPalette*

An EODisplayGroup object maintains associations between values in enterprise objects and user interface controls. It manages a group of enterprise objects and provides in-memory sorting.

An EOEditingContext object is automatically added to your application along with an EODisplayGroup when you drag an entity into the nib file window, as described in “Dragging a Model File into the Window” on page 145. Consequently, you normally don’t need to drag this object off the palette.

EODisplayGroup, Associations, and Class Keys

EODisplayGroups synchronize the data displayed in the user interface with the corresponding data in an enterprise object. An EODisplayGroup:

- Tracks the selection as the user changes it
- Applies updates from user interface objects to enterprise objects
- Applies changes in enterprise objects back to user interface objects

EODisplayGroups use EOAssociations to mediate between enterprise objects and the user interface. An association ties a single user interface object to a value corresponding to a key (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.

The term *class keys* in this context refers to the EOKeyValueCoding informal protocol, in which the properties in an enterprise object are accessed as key-value pairs. An enterprise object class can carry its properties either as instance variables or as an NSDictionary object. In a specific instance of an enterprise object, each key has a corresponding value. For example, an instance of the Movie class has the key **title**, which might have the value "Citizen Kane." An association can access the value "Citizen Kane" through the key **title**. So, for example, if you change the value "Citizen Kane" to "Malcolm X" in a TextField for the key **title**, the association communicates the change back to the enterprise object through the EODisplayGroup.

For more information, see the EODisplayGroup and EOAssociation class specifications in the *Enterprise Objects Framework Reference*.

Creating a New Application

The following sections describe how to use the EOPalette objects, as well as the standard control objects, to create a user interface for an application.

When you open a nib file from your project to start designing your user interface, Interface Builder displays the nib file window.

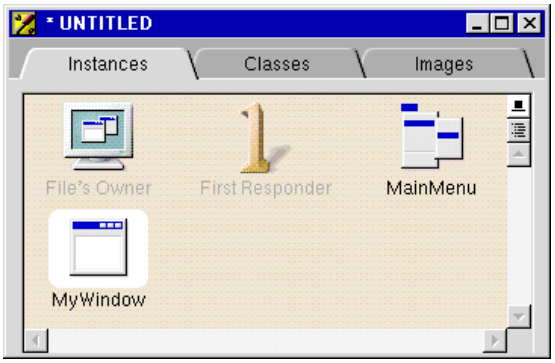

Figure 34. *The Interface Builder nib file window*

Adding EODisplayGroup and EODatabaseDataSource Objects

To display data in your user interface, you need an EODisplayGroup object. EODisplayGroups manage associations between the values of class keys and objects (typically, user interface objects). 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), you drag an entity from EOModeler into either the window for the interface you're building or the nib file window.

The EOPalette provides a plain EODisplayGroup object that can be used to create a detail EODisplayGroup; it can also be used if you want to programmatically set a data source. For more information on using detail EODisplayGroups, see "Creating a Master-Detail Interface" on page 151.

Dragging a Model File into the Window

You use EOModeler to display the entities available for a particular database, and to define the mapping between an entity and an enterprise object class. Once you've defined a model, you can use it to create an entity EODisplayGroup in Interface Builder by dragging in either the model file itself or a single entity. When you drag in the entire model file, Interface Builder displays the Set entity panel, prompting you to select the entity you want to add to your nib.

For more information on using EOModeler to create model files, see the chapter "Using EOModeler."

To create an entity EODisplayGroup from a model file, you can drag an entity from EOModeler into the Interface Builder nib file window, as described in the chapter "Getting Started." However, you can also drag an entity from EOModeler directly into your window. This creates a table view pre-connected to all of the attributes in your entity that you've specified as class properties.

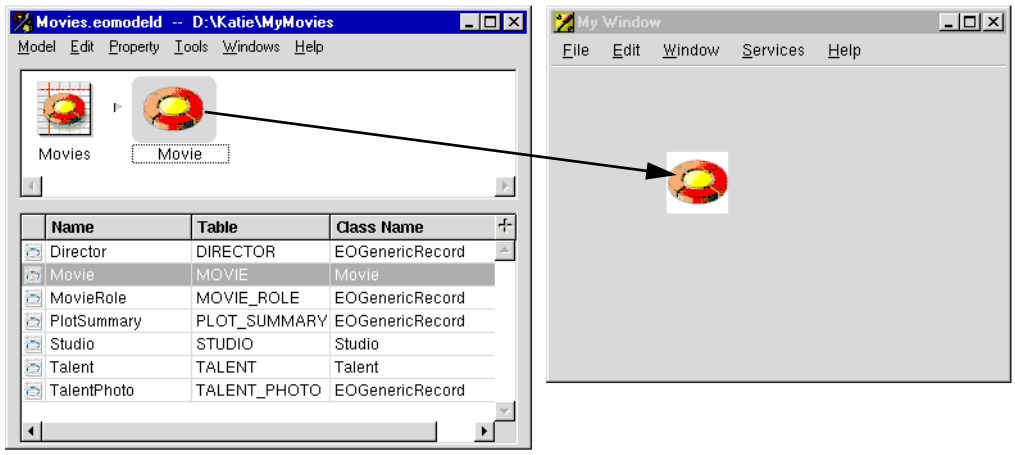

Figure 35. *Dragging an Entity from EOModeler into Interface Builder*

Figure 36 shows the results of dragging an entity into your window. In the nib file window, there's a new entity EODisplayGroup that's named after the entity you dragged in. Note that the nib file window also includes an EOEditingContext object. An EOEditingContext object is added to your application along with the first entity you drag into Interface Builder. Because an application typically only needs one EOEditingContext, this object is only added once.

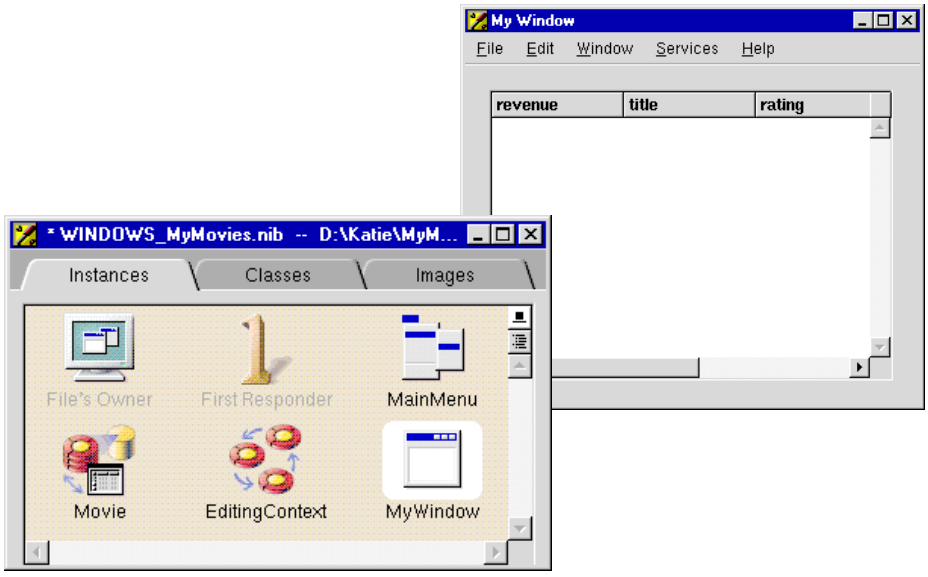

Figure 36. *An Entity EODisplayGroup*

Double-clicking an entity EODisplayGroup in the nib file window launches EOModeler, where you can edit the model that includes the entity. Any edits you make and save to a model file after you drag it into Interface Builder are reflected in the entity EODisplayGroup created from the model.

By clicking the Outline mode button in the nib file window and clicking the button to the left of EODisplayGroup, you can see that the entity EODisplayGroup actually includes both an EODisplayGroup and an EODatabaseDataSource object, as shown in Figure 37. Clicking the button expands and contracts the outline.

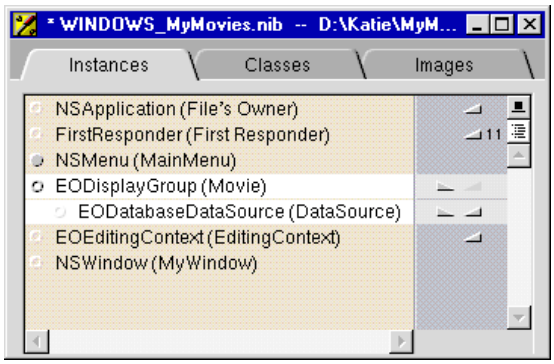

Figure 37. *Examining an Entity EODisplayGroup in Outline Mode*

Inspecting an Entity EODisplayGroup

An entity EODisplayGroup has access to the keys of its associated enterprise object class. These keys correspond to the properties you supplied for the class in EOModeler. For more information on class keys, see "EODisplayGroup, Associations, and Class Keys" on page 143.

The EODisplayGroup Inspector lets you examine class keys and set options used by the entity EODisplayGroup. To display the EODisplayGroup Inspector, select the entity EODisplayGroup in the nib file window and choose Tools m Inspector.

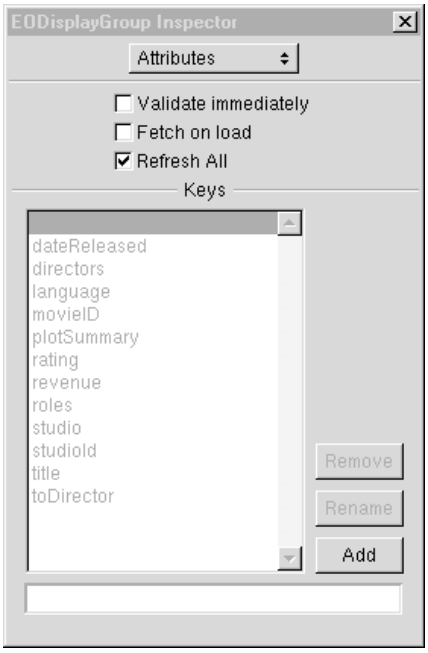

Figure 38. *Inspecting an Entity EODisplayGroup*

Validate Immediately

The Validate immediately checkbox lets you specify that validating should occur at the user interface level, as soon as a user enters a value.

Fetch on load

The Fetch on load checkbox lets you specify that a fetch should automatically be performed for the EODisplayGroup when the nib file is loaded.

Refresh All

The Refresh All checkbox lets you specify whether the EODisplayGroup should refresh all of its EOAssociations when objects change, even if it isn't directly displaying the changed objects.

When objects change in the EOEditingContext for an EODisplayGroup, the EODisplayGroup by default refreshes all of its EOAssociations, even if none of the EODisplayGroup's objects is in the EOEditingContext notification change list.

This "universal" refresh is sometimes necessary because EOAssociations may display derived values (through key paths or business methods) that depend on objects other than the ones being displayed. However, if you know that your

user interface doesn't display any such derived data, you can set your EODisplayGroup to refresh its EOAssociations only if its (the EODisplayGroup's) objects were updated. You do this by unchecking Refresh All.

You can accomplish this programmatically, by using a statement such as the following:

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

You can also implement the EODisplayGroup delegate method displayGroup:shouldRedisplayForChangesInEditingContext: to control when redisplay occurs.

Keys

The Keys area lists the keys for the enterprise object class associated with the entity; keys correspond to the properties you specified for the class in EOModeler. For more information on class keys, see "EODisplayGroup, Associations, and Class Keys" on page 143.

The Keys area provides buttons for adding, renaming, and deleting keys. You can't rename the keys that an entity EODisplayGroup derived from a model. You can use the Add and Rename buttons to add the keys to a non-entity EODisplayGroup for which you have programmatically supplied a data source, or you can add a key that doesn't correspond to a class property to an entity EODisplayGroup. This field is also useful for adding key paths—for more information, see "Using Key Paths" on page 157. To add a new key, type the name in the text field below the list of keys.

More on Associations

When you drag an entity into your window, the table view that's created has pre-connected associations to the EODisplayGroup created from the entity.

When you explicitly make connections to Enterprise Objects Framework objects in Interface Builder, you have two options—either to set an outlet, or to form an association. However, several objects offer more than one possible association, and those associations in turn can have multiple *aspects*. For example, the associations for user interface controls such as text fields, buttons, and radio buttons have the aspects **value** and **enabled**. You can associate **value** with a key in an enterprise object and **enabled** with a method in an enterprise object whose return value determines whether or not that control should be enabled.

The association for a pop-up list, EOPopupAssociation, has a more extensive set of aspects: **titles**, **selectedTitle**, **selectedTag**, **selectedObject**, and **enabled**. For example, imagine that you have two entities: Employee and Department. You can use the Department entity to populate the pop-up list with the names of all of the departments, then use the **selectedObject** aspect to reflect the department of the currently selected employee. In your application, the pop-up list could be used to change the employee's department.

When you make a connection between any two objects in Interface Builder, the Inspector displays all of the possible options. For example, suppose you make a connection from a text field to an EODisplayGroup. In addition to Outlets, you can also set four different types of associations for the connection: EOPickTextAssoc, EOControlAssoc, EOActionInsertion, and EOActionAssociation.

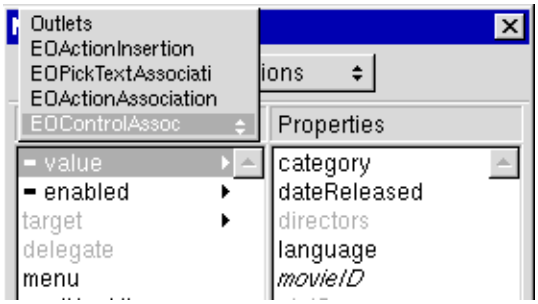

Figure 39. *Associations Available for an NSTextField*

For a description of the different types of associations and their supported bindings, see the EOAssociation documentation in the *Enterprise Objects Framework Reference*.

Note that a single user interface object can only use a single association—switching the pop-up list in the Connections Inspector removes the previous association. For example, you can't specify multiple associations for the connection between a single table column and an EODisplayGroup—associations on a single display object are mutually exclusive. However, an object can be the destination of multiple associations. For example, you can connect different user interface controls to an EODisplayGroup object.

The Connections Inspector uses symbols in the left column and different fonts in the right column to indicate which aspects are appropriate to bind to which class keys. When you select an aspect in the left column, 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. Class keys that aren't class properties are

italicized. The Connections Inspector will let you connect to keys in italics. However, you should avoid doing this unless you have a special reason for connecting to a key that's not a class property (for example, you might want to make a connection to a class method).

The following table describes the symbols associated with aspects in the Connections Inspector:

Symbol	Meaning
	When this symbol appears next to an aspect in the left column of the Connections Inspector, it 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). For example, this symbol appears next to the **value** aspect, which is used to display the value of a particular class key.
	When this symbol appears next to an aspect in the left column of the Connections Inspector, it means that the aspect should be bound to a class key that represents a to-one relationship.
	When this symbol appears next to an aspect in the left column of the Connections Inspector, it means that the aspect should be bound to a class property that represents a to-many relationship.

Creating a Master-Detail Interface

A master-detail presentation is a way of displaying a to-many or a to-one relationship. In this configuration, the master table holds records for the source of the relationship; the detail table contains records for the destination. As individual records in the master table are selected, the contents of the detail table change to show the records that correspond to the selection in the master.

The chapter "Getting Started" describes how to create a master-detail interface by explicitly making connections between EODisplayGroups and table views. However, you can also create a master-detail interface by simply dragging a relationship from EOModeler into your window.

To create a master-detail interface:

1. In EOModeler, create a model that includes the relationship you want to display in a master-detail interface.

For example, in a model based on the Movie database, you can add a to-many relationship called roles from Movie to Role; one movie has many roles. For more information on adding relationships to a model, see the chapter "Using EOModeler."

2. Drag the relationship from EOModeler into your window.

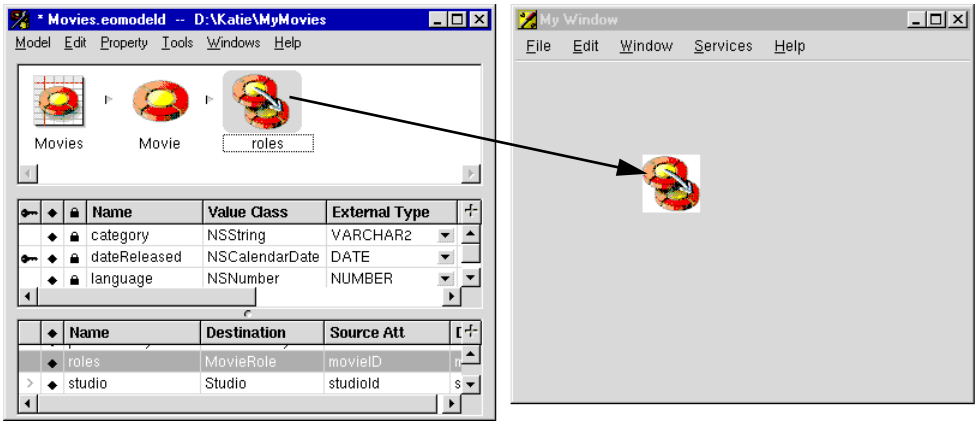

Figure 40. *Creating a Master-Detail Association*

This operation creates a master-detail interface. Columns are automatically added for all of the attributes marked as class properties; you can delete any columns you don't want.

To test the interface, choose File m Test Interface. Note that when you select a movie in the left (master) table, the display in the right (detail) table changes to display the roles in that movie.

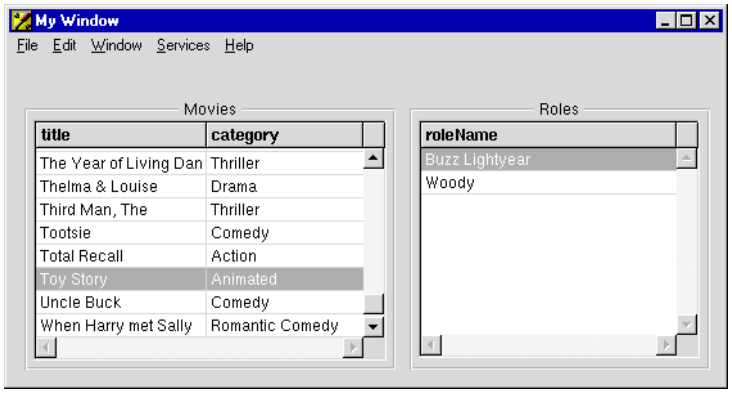

Figure 41. *Master-Detail in Action*

Creating a Master-Peer Interface

You can also create a master-peer interface. In a master-peer configuration, instead of using an entity EODisplayGroup and a plain EODisplayGroup object as you do in a master-detail interface, you drag two entities or models into Interface Builder. Each of the resulting entity EODisplayGroups has its own data source. You then connect the two entity EODisplayGroups in the same manner that you connect an entity EODisplayGroup and a plain EODisplayGroup in a master-detail interface.

Note: In pre-2.0 Enterprise Objects Framework releases, the primary motivation for supporting master-peer configurations was to be able to put a qualifier on the peer. However, this limitation is now largely addressed by the ability to put in-memory qualifiers on EODisplayGroups.

The difference between these two configurations is that master-detail operates directly on the object graph. The master has a class property that represents a to-many relationship to the detail. When you make changes to the detail you're directly modifying the master's relationship array. Enterprise Objects Framework handles relationship manipulation quite well in this scenario.

With master-peer, however, you're not operating directly on the object graph—instead you're going to the database for peer information. The master need *not* have a class property that represents a to-many relationship to the peer. Consequently, when you make changes to the peer, you're not directly modifying the master's relationship array (assuming the master even has the relationship as a class property). If you make changes to the peer the values in the database will be updated, but the master's relationship array (if it exists) won't be.

The upshot is that if you're using a master-peer configuration *and* the master has a to-many relationship as a class property, you're responsible for modifying the master's relationship array to keep it in sync with the peer.

Given this, the scenarios in which you might want to use master-peer instead of master-detail are as follows:

- If the master doesn't have a to-many relationship to the peer defined as a class property
- When the qualifier on the peer can't be executed in memory (for example, because it uses custom SQL or accesses properties not in the object graph)
- When the number of records in the unfiltered set is prohibitively large (so that the filtering is better done in the database)

Using Formatters

The palette includes two formatter objects: 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.

For example, suppose you add fields to a Movie application to display a movie's revenue and release date:

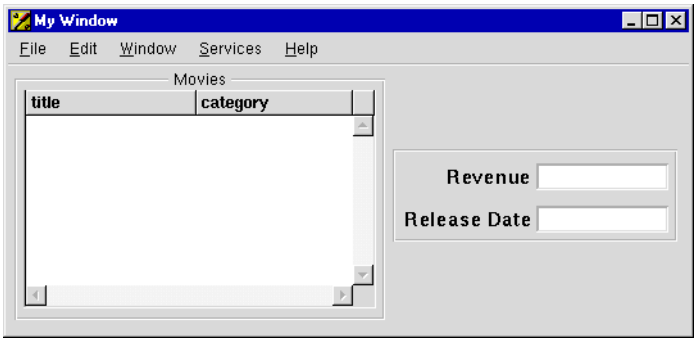

Figure 42. *Adding Fields*

To connect these fields to your Movie entity EODisplayGroup, select each field, control-drag to the EODisplayGroup in the nib file window, and click on the appropriate key in the Inspector (the pop-up list at the top of the left column should be set to EOControlAssociation).

To add custom formatting to these fields, drag the appropriate formatting object from the palette into the field you want to format.

For example, you can drag the Currency formatter into the Revenue text field.

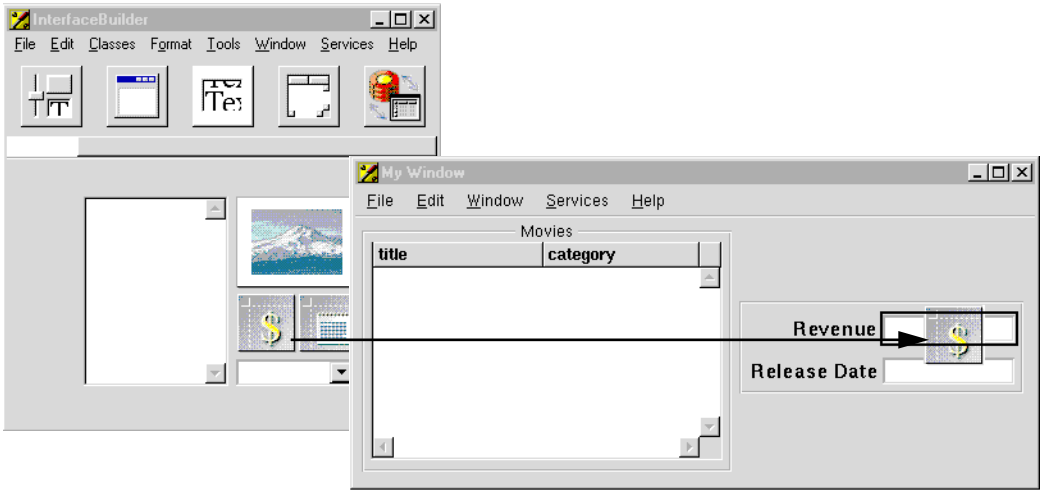

Figure 43. *Dragging the Currency Formatter into a Text Field*

You can then drag the Date formatter into the Release Date field.

You then select a text field and use the Formatter view of the Inspector to change the format for that field, as shown in Figure 44. For example, you can specify that negative values be in red, or enclosed in parentheses.

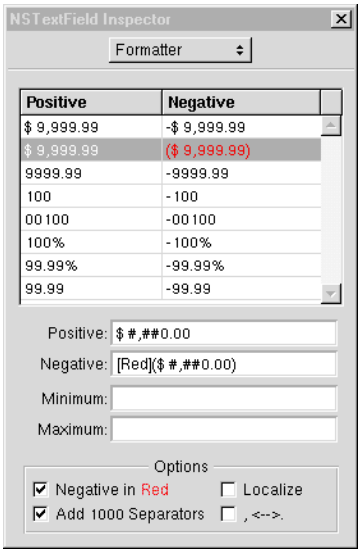

Figure 44. *Specifying a Currency Format*

Likewise, you can change the format of the Release Date field by using the Inspector.

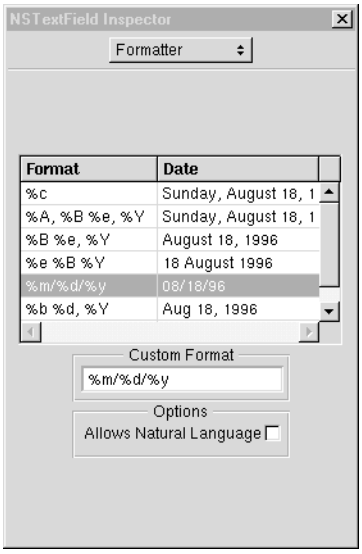

Figure 45. *Specifying a Date Format*

When you enable “Allows Natural Language” for dates in the Inspector, it means that users can change the value of that date field using natural language expressions such as “today,” “yesterday,” “tomorrow,” and so on.

Using Key Paths

The chapter “Using EOModeler” discusses flattening attributes and relationships. An alternative approach to displaying information from one entity in another is to specify a key path in Interface Builder. For example, suppose you want to display the name of the actor who played each role in a movie. You could flatten the actor’s name from the Talent entity into the Role entity in EOModeler. Alternatively, you could specify a key path in Interface Builder. The advantage of using a key path is that unlike flattened properties, which are tied to the database, key paths allow you to traverse the object graph. Because the object graph represents the most current view of data in your application, using key paths is the best way to ensure that your display is always in sync with the data.

To specify a key path:

1. In the nib file window, select the EODisplayGroup object for which you want to specify a key path.

   For example, you can select the “roles” detail EODisplayGroup that represents a relationship between the Movie and Role entities.

2. Display the Attributes view of the EODisplayGroup Inspector.

3. Specify a key path that includes the name of the relationship with the destination table, Talent, and the attribute in that table you want to “add” to roles, and click Add.

   For example, in Figure 47, you can see that the key paths **talent.firstName** and **talent.lastName** have been added to the EODisplayGroup roles. Note that these key paths are both based on a relationship that was defined in EOModeler: **talent**, which is listed among the attributes.

Figure 46. *Specifying a Key Path*

Once you add these keys to your EODisplayGroup object, you can use them in associations just as you would any other key. For example, in the following example application, you can see that the name of the actor is listed alongside the role he or she played in the selected movie.

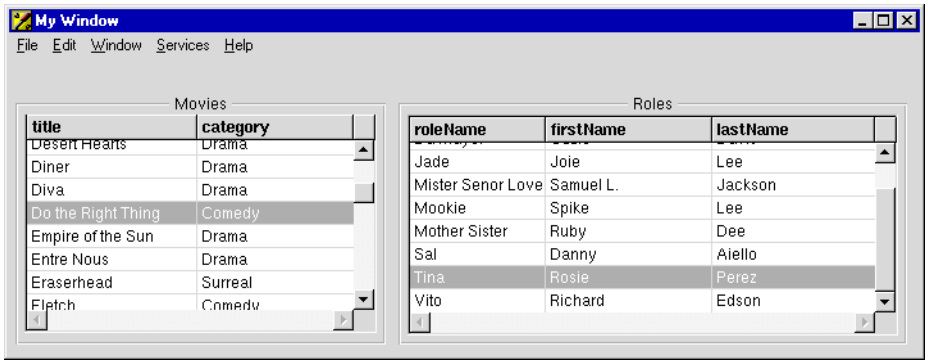

Figure 47. *Using Key Paths in an Application*

Using Different "Document" Configurations

The EOEditingContext object in your nib file can be thought of as representing a single "document"—that is, a particular view of the data.

By default, each nib has its own EOEditingContext (or to put it differently, its own internally consistent view of enterprise object data). Consequently, if you have two different nibs, changes to data in one are only reflected in the other after you save the changes to the database.

You can programmatically modify this default behavior to:

- Use one EOEditingContext for multiple nibs.

  In this scenario, multiple nibs have the same object graph and therefore see each other's changes to objects immediately.

- Use nested EOEditingContexts to construct a "drill down" user interface.

Using One EOEditingContext for Multiple nibs

To set one EOEditingContext for multiple nibs, use the EOEditingContext method **setSubstitutionEditingContext:**. You use this method to substitute the specified EOEditingContext for the one associated with a nib file you're about to load. This method causes all of the connections in your nib file to be redirected to the specified EOEditingContext.

Creating a "Drill Down" User Interface

You can use nested EOEditingContexts to create a "drill down" user interface, in which changes in a nested dialog can be okayed (committed) or canceled (rolled back) to the previous panel. This is possible because EOEditingContext is a subclass of EOObjectStore, which gives its instances the ability to act as EOObjectStores for other EOEditingContexts. In other words, EOEditingContexts can be nested, thereby allowing a user to make edits to an object graph in one EOEditingContext and then discard or commit those changes to another object graph (which, in turn, may commit them to an external store).

To set up a drill down style user interface, use the EOEditingContext method **setDefaultParentObjectStore:**. You use this method before loading a nib file to change the default parent EOObjectStores of the EOEditingContexts in the nib file. The object you supply can be a different EOObjectStoreCoordinator or another EOEditingContext (if you're using a nested EOEditingContext).