Inside WebObjects

Creating a Java Client WebObjects Application

Tutorial

May 2001

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada

Contents

Overview of Java Client

The Java Client feature of WebObjects distributes the objects of an Enterprise Objects Framework application between an application server and one or more clients—typically Java™ applications or Web browsers. It is based on a distributed client-server architecture that uses Java as its development language. This architecture is multi-tier in that processing duties are divided among a client, an application server, and a database server. With a Java Client application, you can partition the business logic and data associated with enterprise objects into a client side and a server side. This partitioning can improve performance and at the same time help to secure legacy data and business rules.

Advantages of Java

To understand the difference that Java makes in client-server architectures, it helps first to consider two of the more common types of client-server applications: the traditional desktop application and the Web application. The two types have complementary strengths and weaknesses.

Characteristic	Desktop Application	Web Application
Interactive	Yes	No
Flexible controls	Yes	No
Rich user-interface paradigm	Yes	No
Portable	No	Yes

Characteristic	Desktop Application	Web Application
Easy to administer	No	Yes
Accessible	No	Yes
Secure	No	Yes

Desktop applications can typically draw upon user-interface frameworks that provide a varied and flexible set of controls, modal dialogs, and multiple windows. On the other hand, HTML has a limited and static set of controls; most of them are forms, active images, and hyperlinks.

A Web application is, by definition, portable since it can run on any client browser that implements certain standards and protocols, regardless of the underlying system; desktop applications are usually limited to the platforms they were built for. Web applications also have high marks for accessibility because they are designed to make it easy for users to get data on networks. Finally, because sensitive data and business logic is confined to the server in Web applications, they tend to be more secure.

Java scores high on each of these characteristics because it can have a strong presence on each side of the client-server divide.The principal advantage of Java is that it runs almost anywhere. The client need only have a compatible Java virtual machine (VM), something that most operating systems and browsers now include as a standard feature. Java applications can be designed to run on the server or the client. Sun's AWT and "Java 2 platform, Standard Edition" (J2SE) packages provide a rich source of flexible, interactive controls for developers.

Thus the promise of Java is the best of both worlds. So what are some distributed multi-tier Java-based architectures popular today?

Client JDBC applications use a fat-client architecture. Custom code invokes JDBC on the client, which in turn goes through a driver to communicate with a JDBC proxy on the server; this proxy makes the necessary client-library calls on the server. The shortcomings of this type of architecture are typical of all fat-client architectures. Security is a problem because the bytecodes on the client are easily decompiled, leaving both sensitive data and business rules at risk. The server has to be open to allow all client operations without being able to control what the client is doing. In addition, such an architecture doesn't scale; it is expensive to move data over the channel to the client.

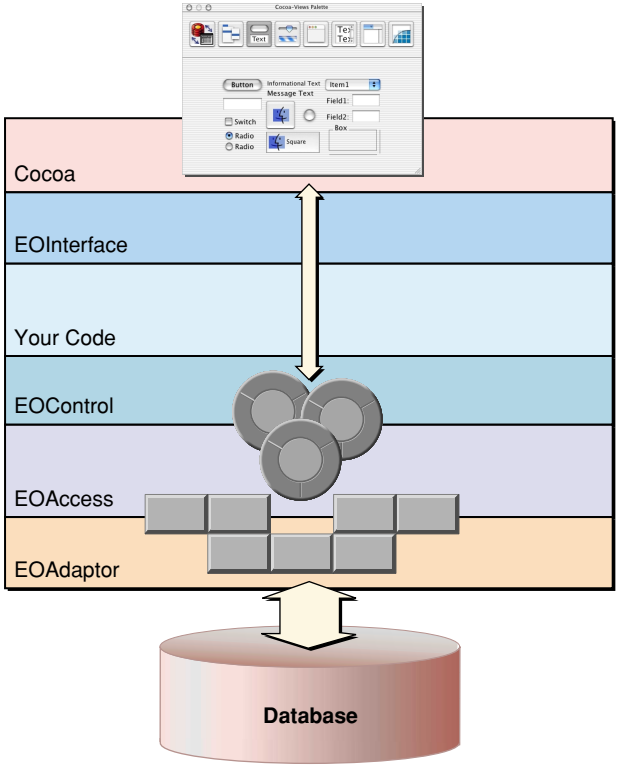
A JDBC three-tier application (with CORBA as the transport) is a big improvement over a client JDBC application. In this architecture the client can be thin since all that is required on the client side is the JFC, nonsensitive custom code (usually for managing the user interface), and CORBA stubs for communicating with the server. Sensitive business logic as well as logic related to database connection are stored on the server. In addition, the server handles all data-intensive computations.

The JDBC three-tier architecture has its own weaknesses. First it results in too much network traffic. Because this architecture uses proxy business objects on the client as handles to the real objects on the server, each client request for an attribute is forwarded to the server, causing a separate round trip and precipitating a message storm. Second, JDBC Three-tier requires developers to write much of the code themselves, from code for database access and data packaging to code for user-interface synchronization and change tracking. Finally, JDBC three-tier does not provide much of the functionality associated with application servers, such as application monitoring and load balancing, nor does it provide HTML integration.

Java Client Architecture

A Java Client application is essentially an Enterprise Objects Framework application distributed across an application server (running a WebObjects application) and one or more client applications or applets. As a starting point, consider the diagram in Figure 1-1, which depicts a traditional desktop Enterprise Objects Framework application.

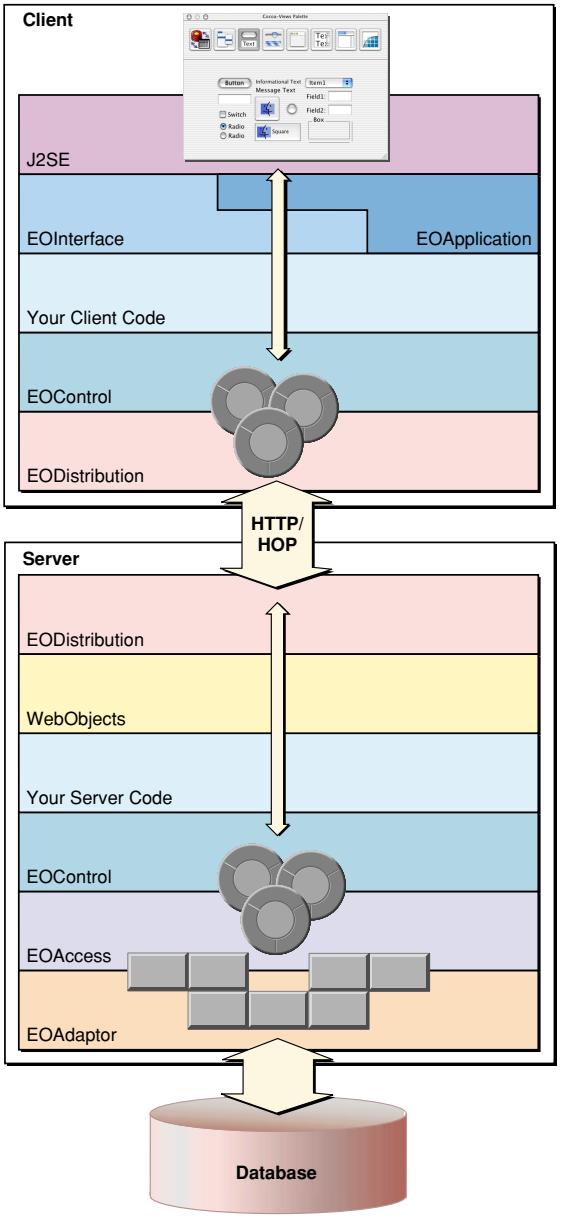
Figure 1-1 Architecture of traditional Enterprise Objects Framework application



In this architecture, data is fetched from databases through the EOAdaptor layer, objects of which (adaptors) interact with specific database servers. The EOAccess layer creates enterprise objects from the raw fetched data and registers these with the EOControl layer; the access layer, through EOModel and related classes, also provides a mapping between the database schema and enterprise objects. The EOControl layer manages a graph of enterprise objects, tracks changes to them, and directs the access layer to commit changes to those objects. Finally, the EOInterface layer in this traditional desktop application synchronizes the data displayed in the user interface—here objects of the Cocoa framework—with the EOControl layer's graph of enterprise objects.

The design of Java Client breaks up some of these layers and redistributes them across the client and the application server, which occupies the middle tier in the overall architecture. Figure 1-2 illustrates how this is done.

Figure 1-2 Java Client architecture



Java Client moves the pieces that perform object-to-user-interface mapping to the client and duplicates the control layer on the client so that the graph of enterprise objects and the management of that graph occurs on both server and client. It also adds a new layer to both client and server, the EODistribution layer, that performs by-copy object distribution and synchronization. The final difference, of course, is the use of J2SE as the user-interface framework. (The object-to-database mapping layer, EOAccess, remains solely on the server.)

The diagram in Figure 1-2 seems to suggest a lot of complexity, but it is important to keep in mind that the functionality it implies (and the amount of code required to implement it) are inherent in all true multi-tier architectures. Java Client provides most of the code for you. Unlike other multi-tier approaches, you do not have to worry about such things as change tracking, data packaging, and user-interface synchronization. In most cases, you need only write your business-logic code.

Data Synchronization Between Client and Server

In a Java Client application, when the user makes a query, the fetch specification is passed through the layers on the client (EOInterface to EOControl to EODistribution), largely through successive invocations of `objectsWithFetchSpecification`. The distribution layer on the client forwards the fetch specification to the server's distribution layer—in the default WebObjects case, synchronously via HTTP. From there the normal mechanisms take over and a SQL call is eventually made to the database server. The database server returns the rows of requested data and, as usual, this data is converted to enterprise objects and is registered with the EOControl layer on the server. The server's distribution layer then sends *copies* of the requested objects back to the client. When the EODistribution layer on the client receives the objects, it registers them with the editing context in the control layer and, through the interface layer's display-group and association mechanisms, the user interface is updated with the requested data.

Although requested objects are copied from the server to the client, and these objects exist in parallel object graphs on both server and client, the enterprise objects on the client usually do not exactly mirror the enterprise objects on the server. The objects on the client usually have a subset of the properties of the objects on the server (although the reverse can also be true). You can partition your application's enterprise objects so that the objects that exist on the client (or the server) have a restricted set of data and behaviors.

Once the client has fetched data, this data is cached and is represented internally by the client's object graph. As users modify the data (or delete or add rows of data), the client's object graph is updated to reflect the new state. When users request that this data be saved, the changed objects are pushed to the server. If the business logic on the server validates these changes, the changes are committed to the database.

Note that Java Client automatically pushes updates from the server to the client. It also, by default, pushes changes before client-side objects remotely invoke methods on server-side objects.

Java Client as a WebObjects Application

Out of the box, Java Client runs as a type of WebObjects application. In the multi-tier architecture described earlier, WebObjects provides an application server as well as HTML and HTTP support. The distribution layer on the client provides an HTTP channel to handle communication between the application server and the Java Client application.

A Java Client WebObjects application gives you considerable flexibility in how you compose the pages of your application. However, it is strongly recommended that your Java Client projects be executed on the client as stand-alone applications instead of applets. By doing this, you avoid the many compatibility issues present when running applets inside different browsers. However, if your business requires that your project use applets on the client instead of applications, you can combine Java Client applets and static and dynamic (WebObjects) HTML elements in various ways. You can have pages with or without Java Clients or pages with multiple Java Clients, each with its own controller. For example, you could have a login page that takes the user to one of many Java Client pages based on some piece of account data. In addition, Java Client applets are not limited to the downloaded JFC components; as can any applet, they can create dialogs and secondary windows on the fly.

When you create a Java Client project using Project Builder, the project is organized using three targets:

- The Web Server target, which lists the files required to build the client application.

This target contains an interface file that stores an archive of J2SE (and other 100% Pure Java) objects. That file's owner is a custom subclass of EOInterfaceController. For more on Interface Builder see "Creating the User Interface" (page 51).

- The Application Server target, which lists the files needed to build the server application.

  This target includes the `Main.wo` component that contains a subcomponent of type WOJavaClientApplet. Also included are the `Application`, `Session`, and `DirectAction` classes.

- The root target, named after the project, which groups the Client and Server targets. It is used to build the entire project (the Client and Server products) as a single unit.

For more on Project Builder's targets see "File Organization in Project Builder" (page 46).

The frameworks required to build a Java Client application are also automatically added to your project.

When you launch the client side of a Java Client application (an application or an applet), the server application creates a WOSession object. It contains the EOEditingContext and EODistributionContext objects used to manage enterprise objects (read from and write to a database) and to synchronize them with their counterparts on the client.

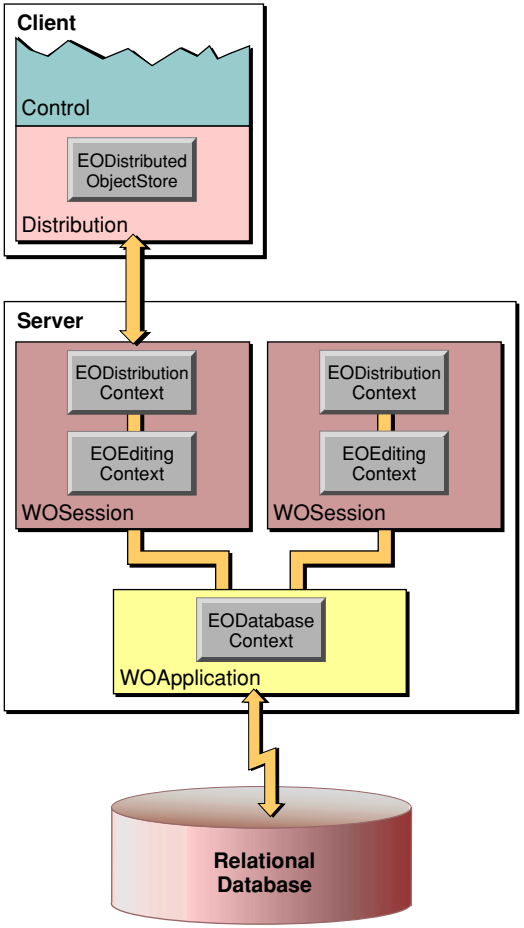
With Java Client applets, the client automatically downloads the classes that it needs to run from the server if they are not already installed on the client's classpath. This way the client-side applet is always updated. There are several disadvantages with this approach:

- **Longer application launch times:** Launch times suffer because the client has to make sure that the appropriate classes are installed before the applet can run.
- **Security restrictions:** Browsers impose restrictions on the actions that downloaded code can perform on the client machine (those restrictions can be modified or removed by editing preference settings in the browser).
- **Application instability:** Java Virtual Machine (VM) implementations in browsers are generally poor. When more than one applet are run inside a browser, they often share the same VM, contributing to application instability.

When the client-side applications are executed as applications instead of applets, launch times are minimized because the server sends only data to the client, not class implementations. However, when a new version of an application is developed, the client needs to be manually upgraded; otherwise, an exception is thrown when the user tries to run the application.

As you can see from the diagram in Figure 1-3, each session created and managed by the WebObjects application has, if it is communicating with a Java Client application, its own editing context and its own server-side distribution layer. As described earlier, communication between the server and client is handled through the distribution layers on the server and the client. The WOApplication maintains the object store (EODatabaseContext) for all its sessions.

Figure 1-3 Java Client in a WebObjects application

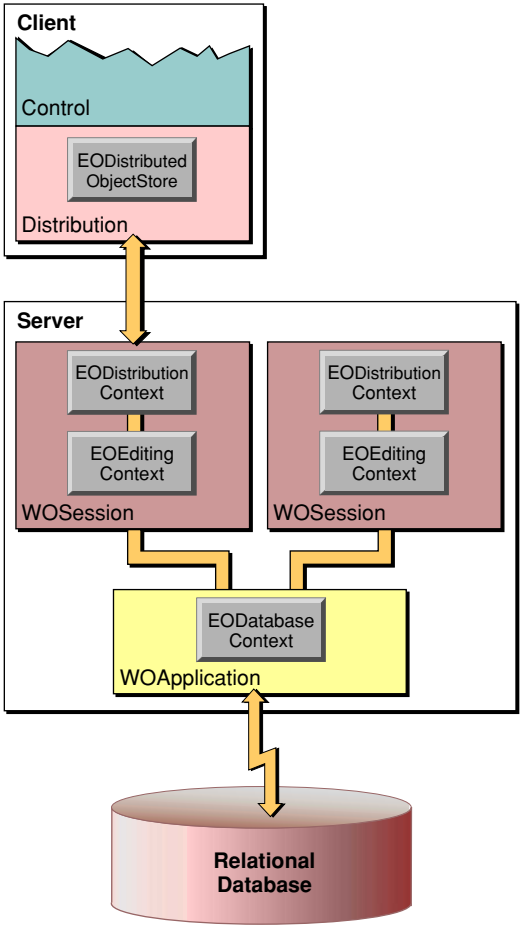


The session object is, by default, the delegate of the distribution layer's EODistributionContext, the object that handles communication on the server. The EODistributionContext class defines several security-related delegate methods for validating remote invocations; if you wish, you can simply implement these methods in your Session class.

Java Client Layers and Classes

The classes specific to Java Client are found in the distribution layers implemented in `com.webobjects.eodistribution` (both client and server) and in the client's control and interface/application layers (`com.webobjects.eocontrol`, `com.webobjects.eoapplication`, `com.webobjects.eointerface` and `com.webobjects.eointerface.swing`). The major classes used in Java Client applications are depicted in Figure 1-4.

Figure 1-4 Major classes in a Java Client application



Client Interface and Control Layers

The interface layer displays to the user the properties of the enterprise objects maintained in the control layer, using display groups and associations. Changes to the object graph are automatically synchronized with the user interface and user-entered data is automatically reflected in the object graph. The primary mechanisms behind this synchronization are display groups (EODisplayGroup) and associations (EOAssociation subclasses).

The interface layer is tightly integrated with the application layer. Java Client applications typically execute as stand-alone applications or as applets running in a browser. The application layer isolates the developer from the idiosyncrasies of each execution environment. It provides classes that are used to manage application-level data and resources, including arguments, defaults (transient and persistent), localization information, menu operations (like Save All and Quit), documents, user interface controls, and so on.

An additional layer, called the generation layer, is added to the client side in Direct to Java Client applications. It is used to dynamically generate user interface elements based on actions performed by the user and rules defined by the developer. For more on Direct to Java Client applications see *Getting Started With Direct To Java Client*.

The EOControl layer's primary responsibility is the management of the object graph through an EOEditingContext. This layer exists on both the client and server side of the application. It also implements faulting (on-demand fetching) and tracks editing changes.

These are the differences between the client and server layers:

- The interface layer on the client is implemented using the J2SE architecture.
- The object store and the data source used by the client control layer are objects in the distribution layer; essentially, these objects communicate changes to the object graph across the channel to the server.
- The client layers include APIs that enable remote invocations of server methods.

The Distribution Layer

The distribution layer is responsible for synchronizing the states of the object graphs on the client and on the application server in the middle tier. The distribution layer moves properties in both directions, that is, as it fetches objects and saves changes.

The distribution layer has a client side and a server side. The client side uses EODistributedDataSource and EODistributedObjectStore classes. The server side uses the EODistributionContext class.

Client Distribution Classes

The client-side distribution layer has several public classes. These are the main ones:

- **EODistributionChannel, EOHTTPChannel:** The distribution layer provides channels through which the application server and the Java clients communicate. The EOHTTPChannel class implements an HTTP channel, that is used by Java Client WebObjects applications, but you can subclass the abstract class EODistributionChannel and implement a channel that uses a different transport protocol (such as CORBA). On the client side EODistributedObjectStore handles communication over the channel; on the server side it's EODistributionContext.
- **EODistributedObjectStore:** On the client the distribution layer provides a distributed object store. It handles interaction with the distribution layer's channel (an EODistributionChannel object), incorporating knowledge of that channel so it can forward messages it receives from the server to its editing contexts and forward messages from its editing contexts to the server.
- **EODistributedDataSource:** A concrete subclass of EODataSource (defined in EOControl) that fetches using an EOEditingContext as its source of objects; the editing context, in turn, forwards the fetch requests to its object store (usually an instance of EODistributedObjectStore) where it is ultimately serviced by an EODatabaseContext on the server.

Server Distribution Class

The server-side distribution layer has the EODistributionContext class. It encodes data to send to the client and decodes data it receives from the client over the distribution channel. It also keeps track of the state of the server-side object graph so it can communicate any changes to the client and thus synchronize the object graphs. The EODistributionContext (or its delegate) also validates remote invocations originating from client objects.

Other Server Elements

An additional server element used in the distribution layer of the server is the WOJavaClientApplet component. It is used to download and create an applet of class `com.webobjects.eoapplication.EOApplet`. It has a dozen or so potential bindings, some general to applets (such as codebase and size) and others specific to Java Client (such as distribution-channel class and interface-controller class). These

bindings are used whether the client is executed as an applet or an application. Therefore, a Java Client application is always configured through a WOJavaClientApplet component.

Programming With Java Client

Generally, programming a Java Client WebObjects application requires some skills and knowledge common to both Enterprise Objects Framework and WebObjects programmers. However, it also requires a specific design technique: object partitioning.

Objects on the server and the client can be instances of custom classes or generic enterprise objects (EOGenericRecord). Objects that derive from custom subclasses can have different sets of properties on both the server and the client. Usually, client objects have the more restricted set of data and behaviors, but it is really up to you to decide based on the requirements of the application and your business. As noted earlier, the primary criteria for partitioning are performance and security. For more on object partitioning see "Getting Your Project Ready to Receive Custom Classes" (page 93) and "Distributing Business Logic in Java Client Applications" (page 99).

Creating the Sample Application

This chapter shows you how to create a Java Client WebObjects application, a distributed Enterprise Objects Framework application. The application is distributed in the sense that business logic can be shared among enterprise objects on the Web client and enterprise objects on the server. The steps you take to create a Java Client WebObjects application are remarkably similar to the steps you take to create a typical stand-alone (or fat-client) Enterprise Objects Framework application.

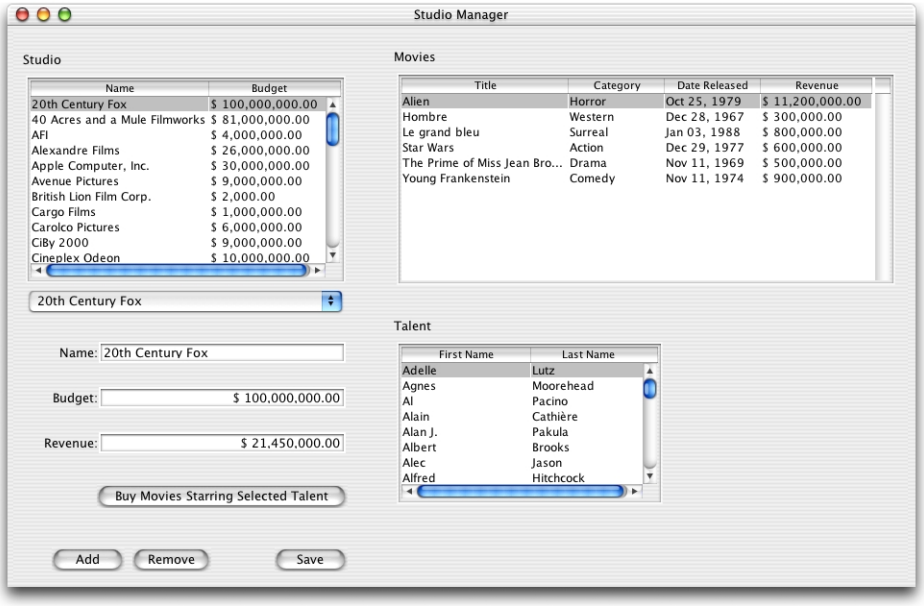
In this tutorial you'll learn the basic things you must do to create a Java Client WebObjects application. You'll discover how to

- create a model of the database using EOModeler
- create a new project using Project Builder
- edit your project's interface file in Interface Builder
- build your project in Project Builder
- write source code for custom enterprise object classes

The Studio Manager Application

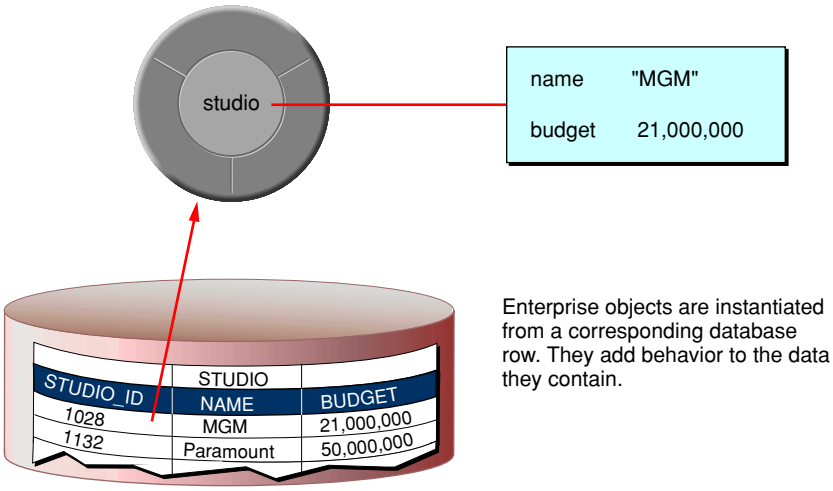
The application you'll be creating in this chapter, Studio Manager, is based on the WOMovies sample database distributed with Enterprise Objects Framework (you must have the sample databases and models installed to do this tutorial). It centers around three types of enterprise objects: Studio, Movie, and Talent. Studios own movies, and they have a budget for buying new movies. Movies feature actors, or

talent. The Studio Manager 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.



Enterprise Objects and Relational Databases

The Studio, Movie, and Talent enterprise object classes correspond to tables in a relational database (WOMovies). 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 the application, Studio objects are instantiated using the data from a corresponding database row, as shown in the following figure:



The enterprise objects in your application do not merely form 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.

In Java Client WebObjects applications, Enterprise Objects Framework manages the interaction between the database (on the server), your enterprise objects (on the server and client), and the user interface (on the client). Its primary responsibilities are as follows:

- fetching data from relational databases into enterprise objects (on the server)
- binding data in enterprise objects to the user interface (on the client)
- keeping objects in the application synchronized with each other, with the database, and with the user interface; this includes keeping enterprise objects on the client synchronized with their counterparts on the server

What Goes Into the Studio Manager Application

As with most Java Client WebObjects applications, you create the Java Client Studio Manager application using the following ingredients:

- **A model of the WOMovies database:** A model defines a mapping between your enterprise objects and data in a relational database. See "What Is a Model?" (page 128) for additional information.
- **A user-interface:** You use Interface Builder to construct a Swing/J2SE-based user-interface that can interact with the user. In Interface Builder's Preferences, you must load a special palette (`EnterpriseObjects.palette`) located in `/Developer/Palettes`. With Interface Builder you can compose a user interface made from "widgets" derived from the Java Foundation Classes (JFC), informally known as Swing.
- **Web Components:** The Main component is automatically set up to have a WOJavaClientApplet component that is bound to the interface controller on the client. You can add other Web components with or without a Java Client linkage. Also provided are skeletal implementation files for the server-side application, session, and direct-action objects as well as API bindings files for server-side components.
- **Source code for custom enterprise object classes:** In the Studio Manager 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 Studio Manager application requires a database server on which you've installed the WOMovies example database. The final ingredients in the application are the Enterprise Objects Framework, WebObjects and Foundation classes, interfaces, and protocols, which you link into your application.

See "What Is an Enterprise Object?" (page 127) for more information on enterprise objects.

Creating the Movies Model

A model file provides a level of abstraction from the database. This enables you to write WebObjects applications that are not tied to a particular database implementation. To create a model file, you need to provide the EOModeler application connection information so that it can communicate with your database. In addition, you define the entities that make up your data model and map them to tables in your database. In EOModeler you also define the relationships between entities using the primary and foreign keys of the database tables. This section guides you through the steps required to accomplish all of these objectives.

If you don't want to learn how to create a model or if you're going to use an existing model, you can go to "Creating the StudioManager Project" (page 39).

Using The EOModeler Wizard to Create the Model

1. Launch EOModeler.

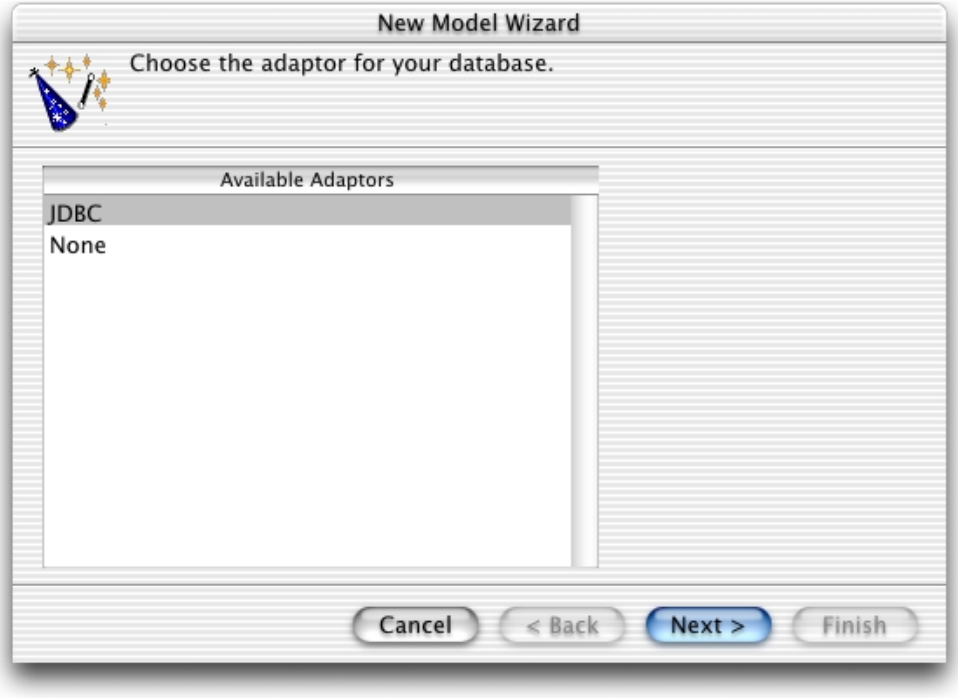
   EOModeler is located in the `/Developer/Applications` directory. Navigate to that directory and launch EOModeler.

2. Create a new model.

   Choose Model > New.

   Ensure that the JDBC adaptor is selected.

   Click Next.

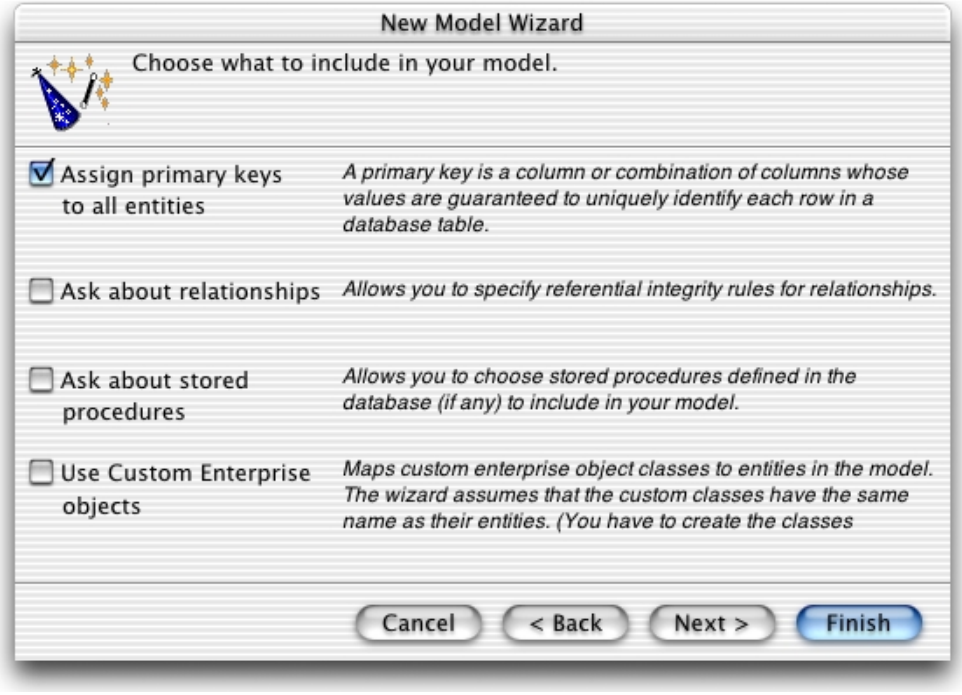


3. Set up the database connection.

   For this tutorial the only information required is the URL. Enter `jdbc:openbase://localhost/WOMovies` in the URL field.

   Click OK.

4. Specify what to include in your model.



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

The basic model the wizard creates contains entities, attributes, and relationships. An entity is the part of the database-to-object mapping that associates a database table with an enterprise object class. For example, the Movie entity maps rows from the MOVIE table to Movie objects. Similarly, an attribute associates a database column with an instance variable. For example, the `title` attribute in the Movie entity maps the TITLE column of the MOVIE table to the `title` instance variable of Movie objects.

A relationship is a link between two entities that's based on attributes of the entities. For example, the Movie entity has a relationship to the MovieRole entity based on the `movieId` attribute of each entity (although the attributes in this example have the same name in both entities, they don't have to). This relationship makes it possible to find all the movie roles in a movie.

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

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

Select the "Assign primary keys to all entities" option.

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

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

Deselect the "Ask about relationships" option.

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

Deselect the "Ask about stored procedures" option.

Selecting this option causes the wizard to read stored procedures from the database's schema information, display them, and allow you to choose which to include in your model. Because the Studio Manager application doesn't require the use of any stored procedures, you don't need this option selected.

Deselect the "Use Custom Enterprise objects" option.

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

If you don't select the "Use Custom Enterprise objects" option, the wizard maps all your database tables to EOGenericRecord. Otherwise, the wizard maps all your database tables to custom classes. The wizard assumes that each entity is to be represented by a custom class with the same name. For example, a table named MOVIE has an entity named Movie, whose corresponding custom class is also named Movie. Use a custom enterprise object class only when you need to add business logic; otherwise use EOGenericRecord. See "Specifying Custom Enterprise Object Classes" (page 91) for more information on enterprise objects.

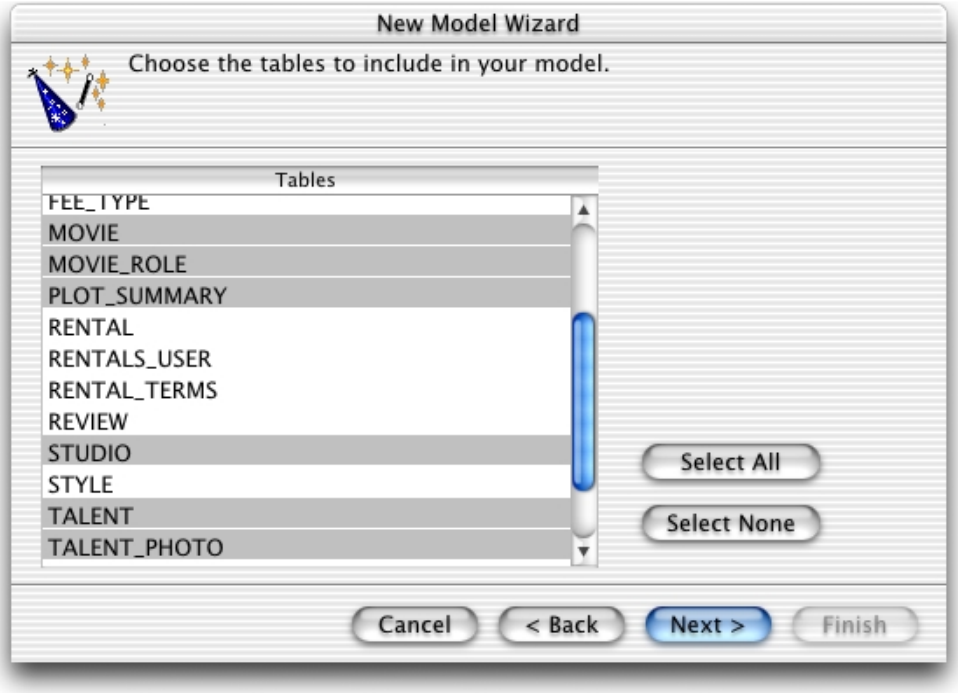
Click Next.

5. Choose the tables to include in your model.

   For this tutorial the only tables required are MOVIE, MOVIE_ROLE, PLOT_SUMMARY, STUDIO, TALENT, and TALENT_PHOTO.

   Select only the tables mentioned above. You need to use the Command key to make a noncontiguous selection.

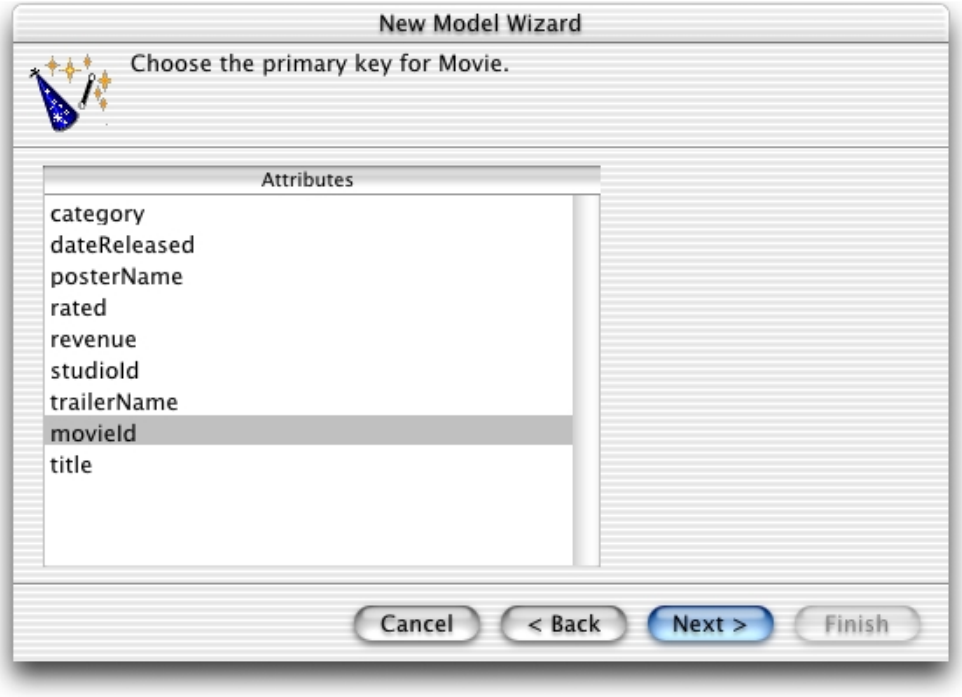
   Click Next.



6. Select a primary key for the Movie entity.

   Select movieId from Attributes list.

   Click Next.

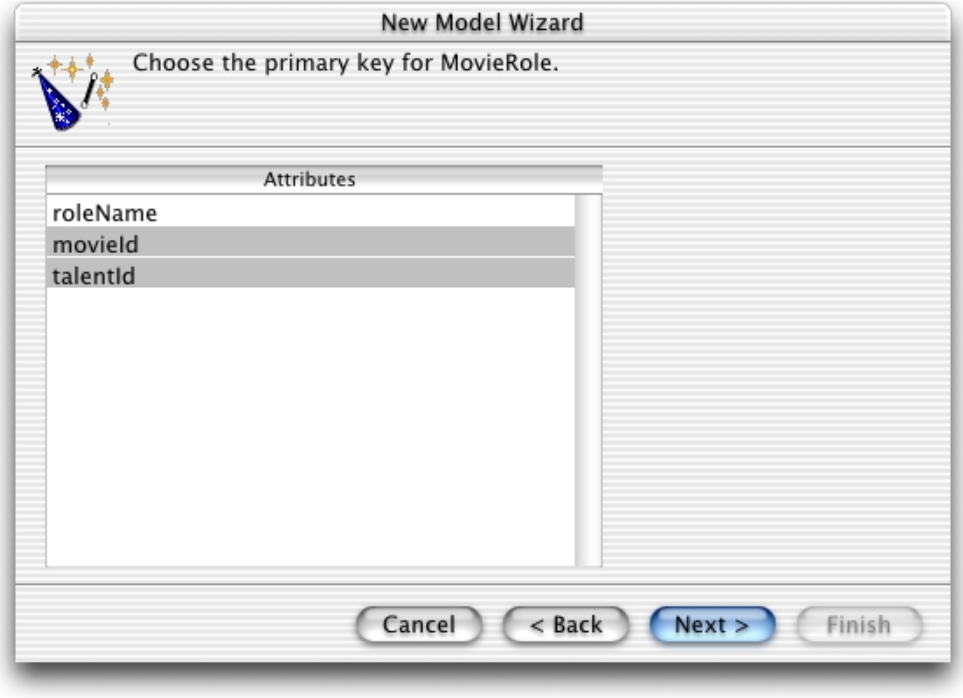


7. Select a primary key for the MovieRole entity.

   The MovieRole entity has a compound primary key: `movieId` + `talendId`. Therefore, the movieId and talentId attributes need to be selected.

   Select the movieId and talentId attributes.

   Click Next.



8. Select a primary key for the PlotSummary entity.

   Select the movieId attribute.

   Click Next.

9. Select a primary key for the Studio entity.

   Select the studioId attribute.

   Click Next.

10. Select a primary key for the Talent entity.

    Select the talentId attribute.

    Click Next.

11. Select a primary key for the TalentPhoto entity.

    Select the talentId attribute.

    Click Next.

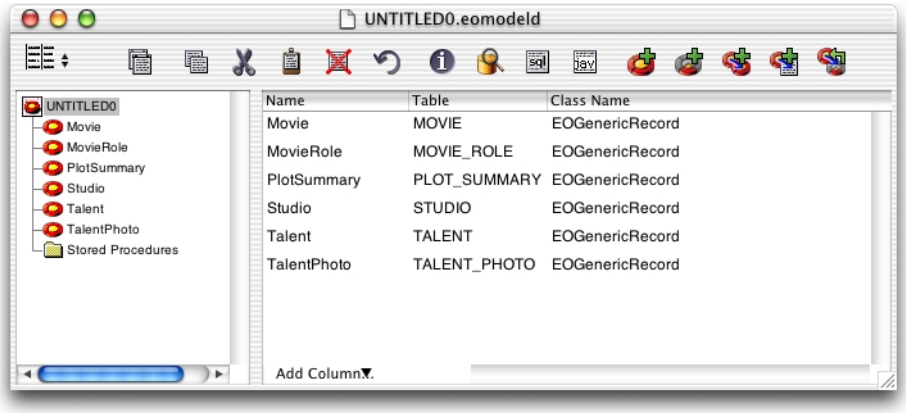
12. Create the model.

    The wizard is done collecting information. It is now ready to create the model.

    Click Finish.

    You should now see a window similar to the one in Figure 2-1.

Figure 2-1 Model generated by the EOModeler wizard

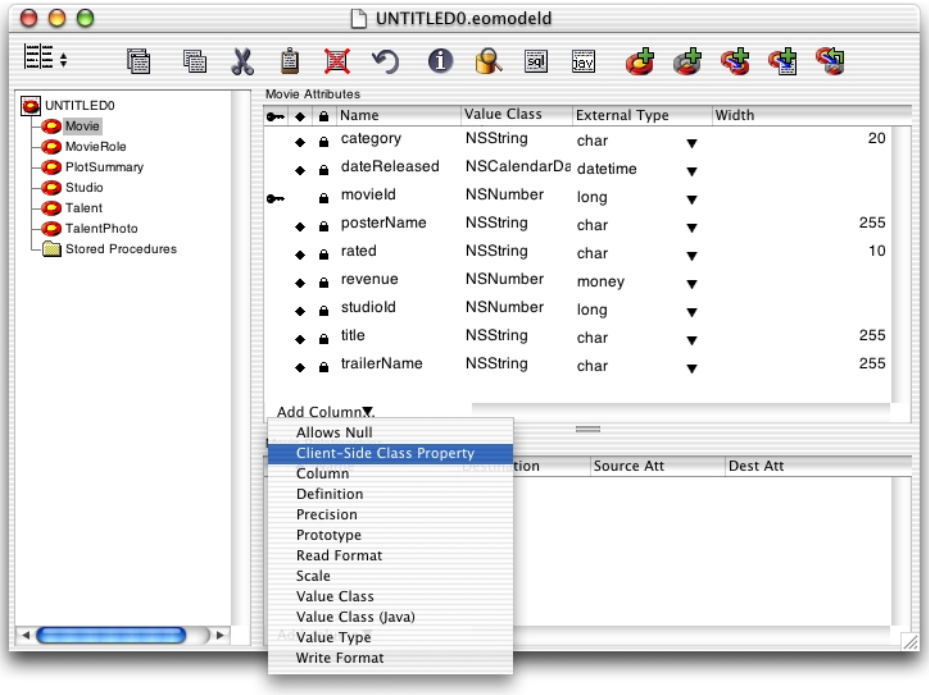


13. Display the client-side class property indicator for attributes.

    Select the Movie entity in the entity browser.

    If you do not see a column with the ⇄ heading in the Movie Attributes table view, you need to add it.

    Choose Client-Side Class Property from the Add Column pull-down menu in the Movie Attributes table view.



14. Display the client-side class property indicator for relationships.

    If you do not see a column with the ⇄ heading in the Movie Relationships table view, you need to add it.

    Choose Client-Side Class Property from the Add Column pull-down menu in the Movie Relationships table view.

15. Save the model.

    Choose Model > Save.

    You are presented with a Save As dialog. Name your model Movies.

    Navigate to a directory where you want to store the model.

    Click Save.

    Close the model window.

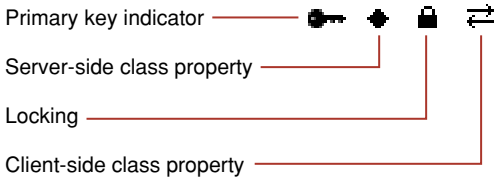
Removing Class-Property Status From Foreign Keys

By default, EOModeler makes class properties for all of an entity's attributes (except for non-database attributes that you add to the entity). When an attribute is a class property, it means that the property is 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 mark as class properties only 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.

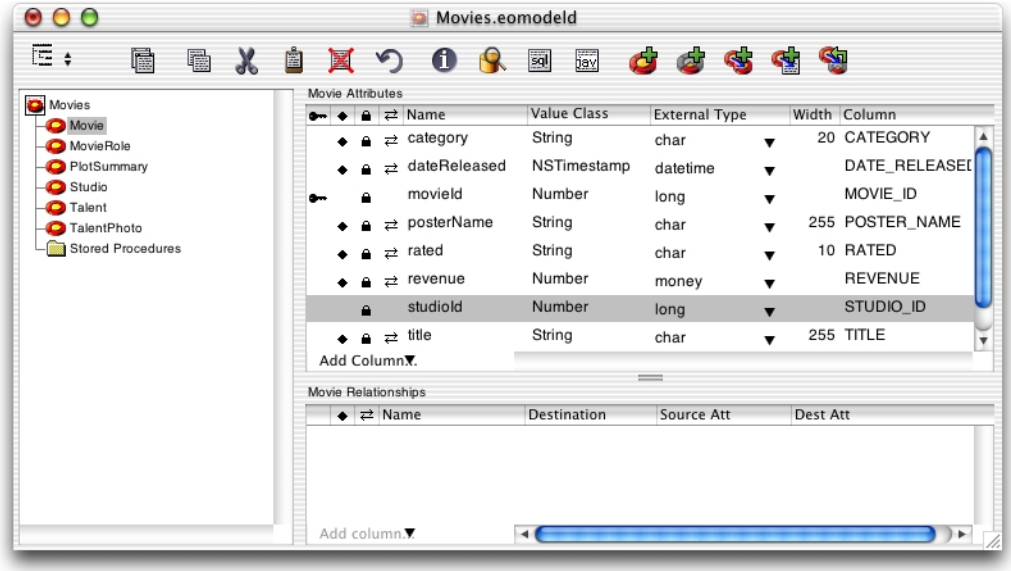
Eliminating primary and foreign keys as class properties has no adverse effect on how Enterprise Objects Framework manages enterprise objects in your application.

EOModeler indicates that an attribute is a server-side class property with the ◆ icon. Client-side class properties have the ⇄ icon.



To remove the class property status from the Movie entity's studioId attribute, follow these steps:

1. Choose Tools > Table Mode.
2. Select the Movie entity.
3. Click the ◆ and ⇄ icons in the `studioId`'s row in the Movie Attributes table, so that they disappear.



Close the model.

The model file you just created will be used as a template when you create the StudioManager project in "Creating the StudioManager Project" (page 39). When you add the model to the project, Project Builder will store a copy of it in the project's directory. From then on, you'll be working with StudioManager's version of the model file. This way, you can create a master model of your database and, from it, generate customized versions for each project you develop.

Creating the StudioManager Project

Every Java Client 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, frameworks, libraries, packages, the application's user interface, sounds, and images. You use the Project Builder application to create and manage projects.

1. Start Project Builder.

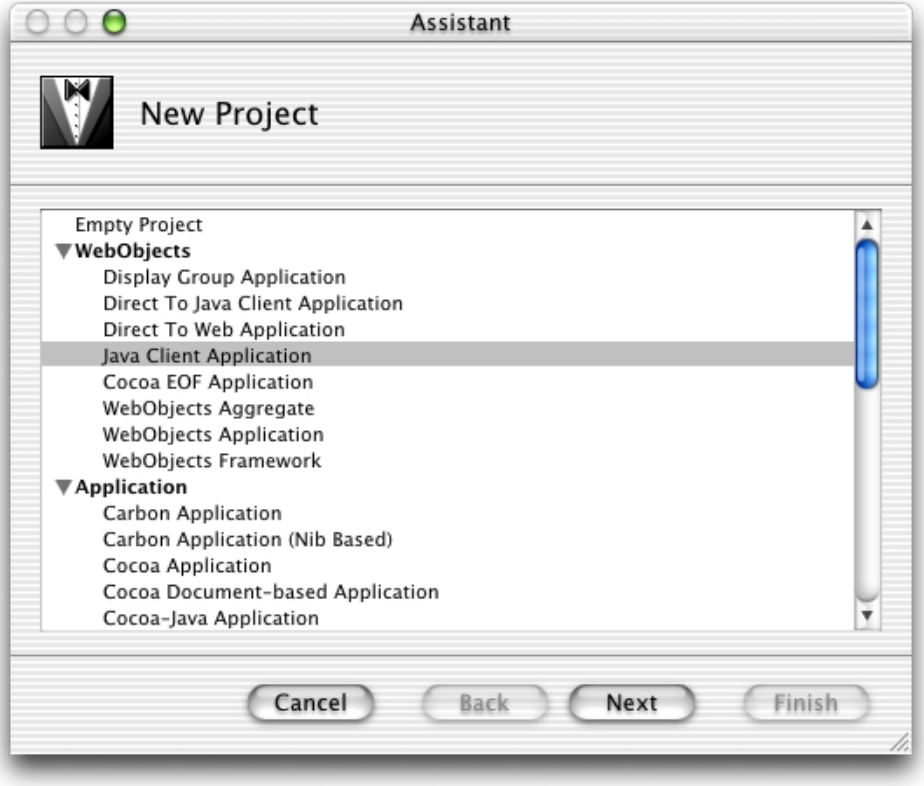
Navigate to `/Developer/Applications` and launch Project Builder.

2. Start a new project.

   Choose File > New Project

   Select Java Client Application.

   Click Next.

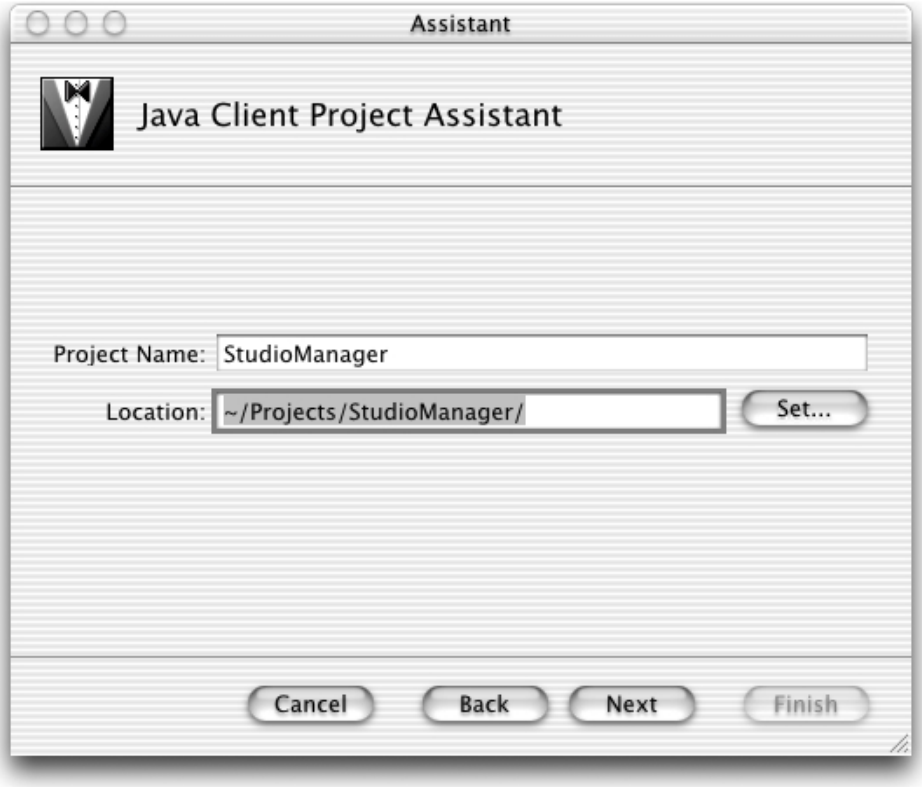


3. Name the project.

Name the project StudioManager.

Click Set and select the folder where you want the project placed.

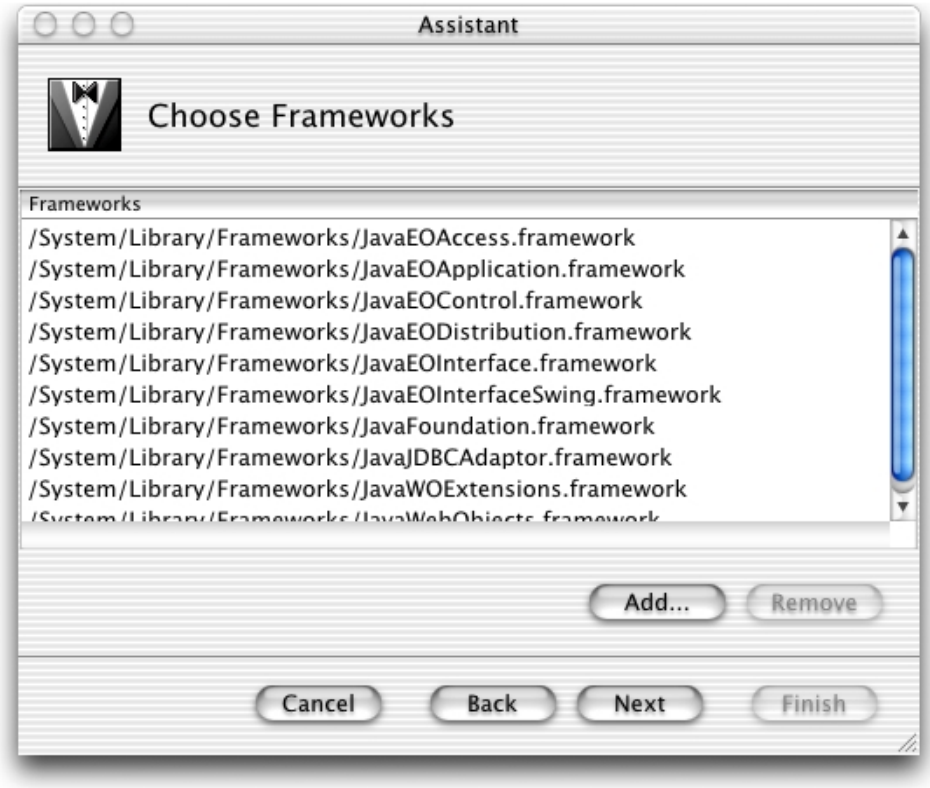
Click Next.



4. Add the necessary frameworks to the project.

   The Choose Frameworks pane allows you to add frameworks to your project, but no additional frameworks are required for this tutorial.

   Click Next.



5. Choose the model to be used in the project.

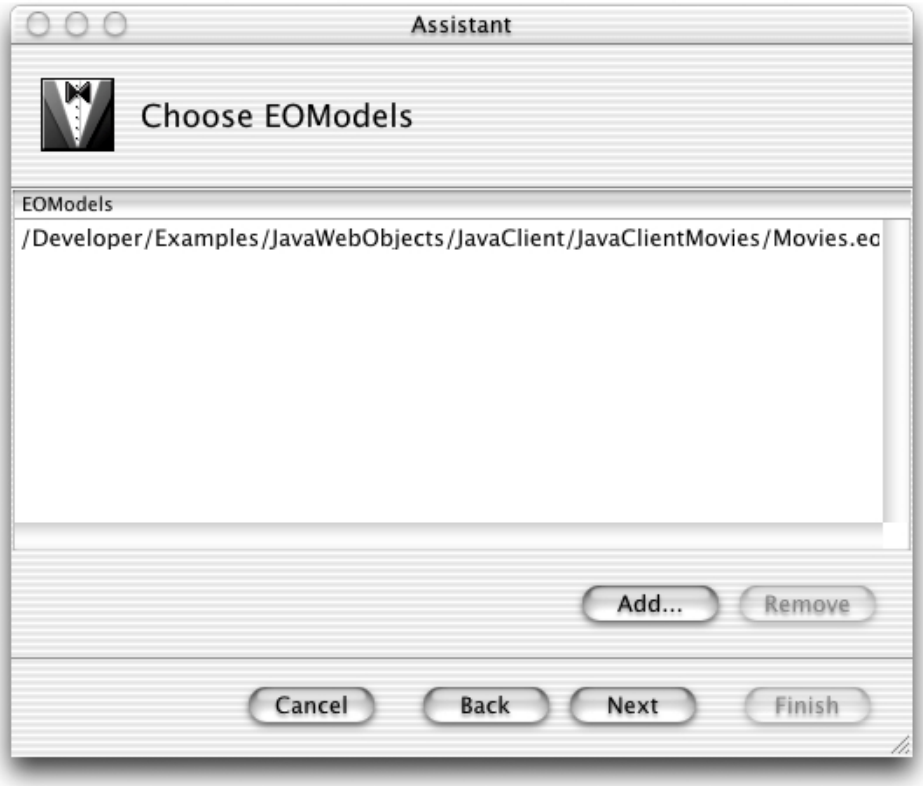
   The Choose EOModels pane allows you to add the model to be used in your project.

   Click Add.

   If you defined your own Movies model, navigate to the folder where you stored it. Otherwise, you can use the Movies model included in one of the example projects. Navigate to `/Developer/Examples/JavaWebObjects/JavaClient/JavaClientMovies`.

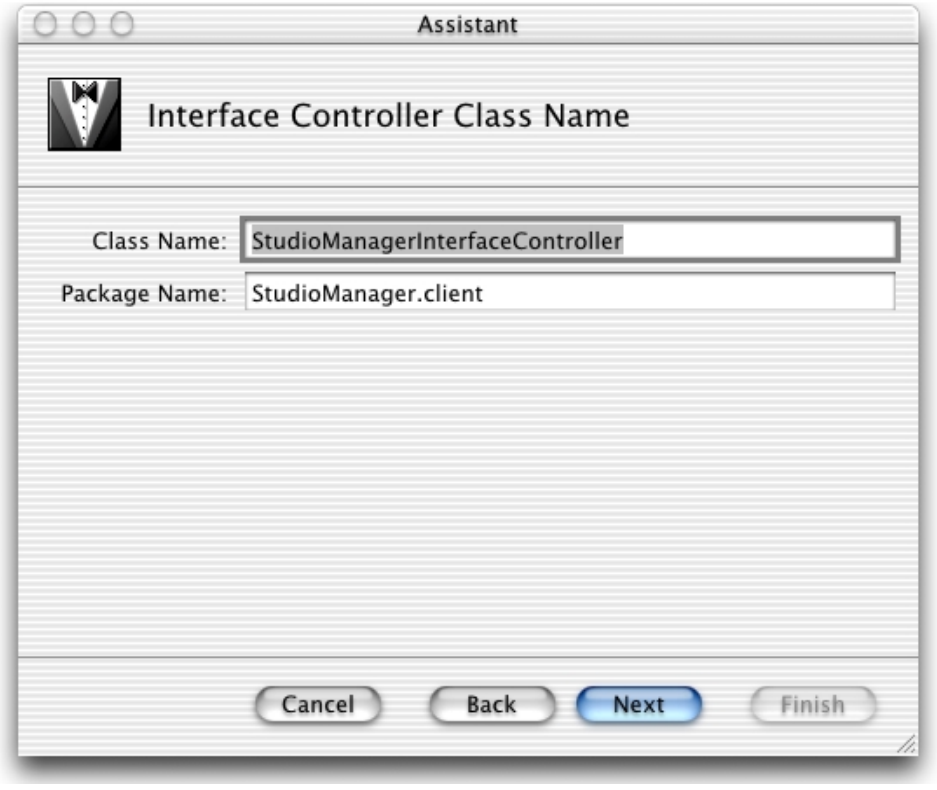
   Select `Movies.eomodeld` and click Choose.

   Click Next.



6. Name the interface file.

   The Interface Controller Class Name pane lets you change the class name and package name of the interface controller. You'll use the default name so click Next.

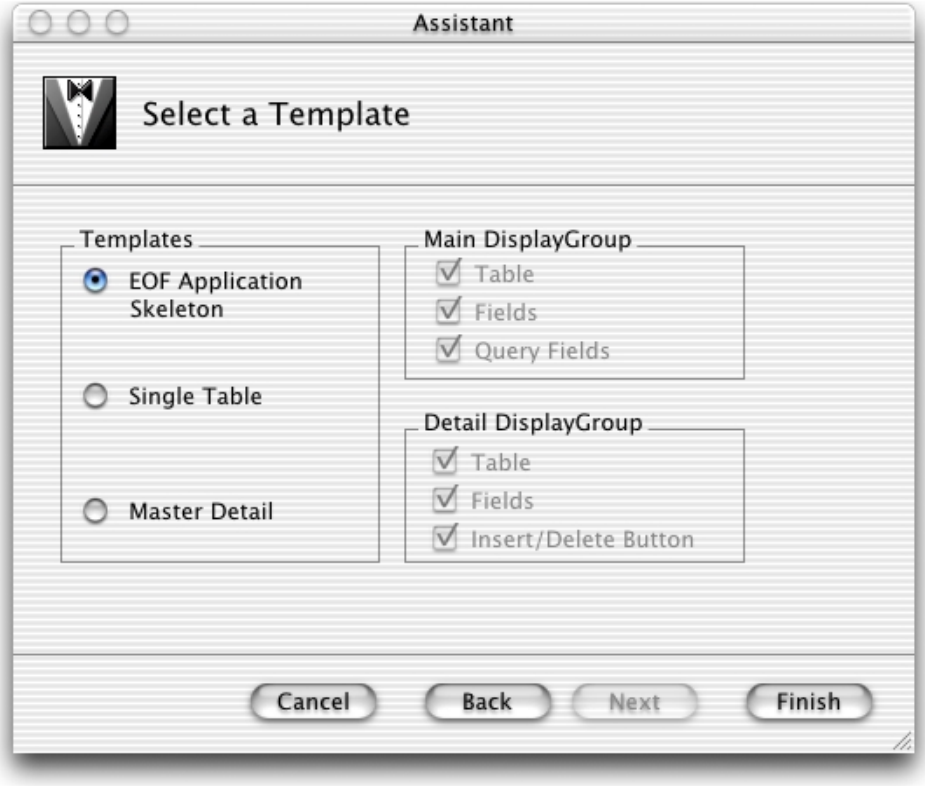


7. Select the application skeleton template.

The Select a Template pane allows you to choose the application template to use for your project.

Make sure that the EOF Application Skeleton template is selected.

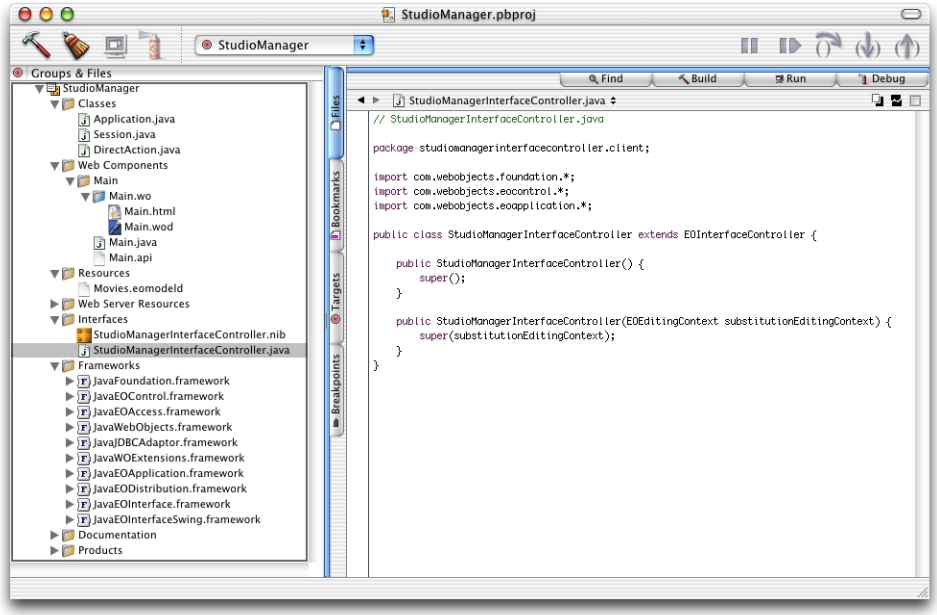
Click Finish to create the project.



The Ingredients of a Java Client Project

Once you've finished with the assistant, Project Builder creates a project directory named after the project—in this case StudioManager—and populates this directory with an assortment of ready-made files and directories. Figure 2-2 shows Project Builder's main window.

Figure 2-2 Project Builder's main window



File Organization in Project Builder

Project Builder offers several organization tools that allow you to visually organize all the files a project requires. This allows you to easily locate a project's files in a central repository. It also lets you assign files to specific targets to facilitate the building process.

Groups

A group is a collection of related files, similar to folders or directories in a file system. They allow you to collect all of your project's components, resources, classes, frameworks, and even other groups under general categories. However, you are not restricted on the type of file you can put in a group.

When you create a Java Client application, Project Builder creates a default hierarchy with seven major groups. You can modify this organization by adding, removing, or deleting groups, and by moving files between groups:

- **Classes** stores the general Java files (`.java`) that the project uses, such as `Application.java`, `Session.java`, and `DirectAction.java`. The Java files related to components, such as `Main.java`, are located in subgroups of the Web Components group.
- **Web Components** stores the dynamic elements used in your project. In a Java Client application, it only has a Main subgroup that contains the `Main.wo` component and related files.
- **Resources** stores the model files (`.eomodeld`) used in the project.
- **Web Server Resources** contains image files (GIF, JPEG) and others that your Web server uses to render a Web page. For example, you can place a GIF file with your business logo in this group.
- **Interfaces** stores the interface files (`.nib`) needed by the client application.
- **Frameworks** stores the framework files (`.framework`) used by your project.
- **Documentation** is where documentation files are placed.

In this scenario, you can think of StudioManager as the root group, because it contains all other groups.

Targets

When a Java Client application is built, two products are created: The client product and the server product. The client product is the client-side application, while the server product is the server-side application. Targets come in two flavors: Build targets and root targets:

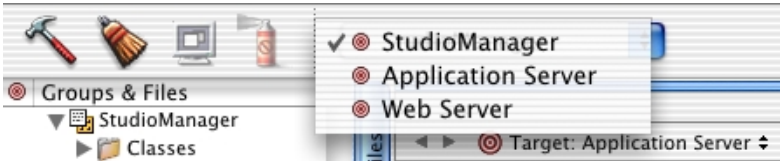
- **Build targets** are used to configure the settings for the corresponding product. When you define a build target, you tell Project Builder which files are part of the target, and how to build its corresponding product.
- **Root targets** are used to group two or more build targets into a single unit; they contain no files or build settings of their own. When an aggregate target is built, the build targets that it contains are built in turn.

These are the targets of a Java Client application:

- The client-side target called Web Server. It contains the files needed to build the client application and the settings used to produce it.
- The server-side target, called Application Server. It contains the files and build settings to be used to generate the server application.

- The root target, named after the project, is used to group the Web Server and the Application Server targets into one unit.

To switch between a project's targets, you use the target pop-up menu.



Client Files

The files associated with the Web Server target are the Interface Builder archive (`.nib`) file and the interface controller (`.java`). The files are named after the project plus "InterfaceController" by default; you can choose a different name when the project is created. In the Interfaces group you will see `StudioManagerInterfaceController.nib` and `StudioManagerInterfaceController.java`.

The Nib File

The nib file in a Java Client application is similar to a nib file in a stand-alone WebObjects application. When you create a user interface, information is added to the Java Client nib file so that it can be used to generate Swing interface elements. This information is encapsulated in a Java archive that is loaded onto the client.

The Interface Controller

In a Java Client application, an interface controller—an EOInterfaceController object—mediates between the user interface and the model objects on the client. When you use Project Builder to create a Java Client project, it automatically generates code for a custom EOInterfaceController subclass and makes an object of this class the owner of the nib file.

In the Model-View-Controller design paradigm, the interface controller plays the role of controller. It has four outlets:

- to its `component`, which is preset to the window in the nib file and functions as the "view" (it can be set to something else)

- to the client's editing context (`editingContext`), which serves as the "model"
- to the controller display group (`controllerDisplayGroup`), a kind of general-purpose display group that contains the interface controller itself and nothing else; through it applications can specify user-interface dependencies and can control the interface through associations
- to the display group (`displayGroup`) in master-detail interfaces

Server Files

The server-side project files created by Project Builder are distributed across several groups. Most notable of these is the Main component (`Main.wo`) in the Main subgroup located in the Web Components group. The `Main.html` file contains this generated HTML code:

```
<HTML>
<HEAD>
  <TITLE>Main</TITLE>
</HEAD>
<BODY>
  <CENTER><WEBOBJECT NAME=Applet></WEBOBJECT></CENTER>
</BODY>
</HTML>
```

The WOJavaClientApplet Component

The `<WEBOBJECT NAME=Applet>` tag in the HTML code above represents a WOJavaClientApplet component. Java Client applications use this component to create an applet (of class `com.webobjects.EOApplet`) and to pass this applet several parameters, some standard, such as size and codebase, and others specific to Java Client applications, such as channel class and interface-controller class.

The `Main.wod` file contains the following default bindings for the WOJavaClientApplet component:

```
Applet: WOJavaClientApplet {
    height = 512;
    width = 512;
    interfaceControllerClassName =
"studiomanagerinterfacecontroller.client.StudioManagerInterfaceController";
    useJavaPlugin = YES;
```

```
}
```

Note that Project Builder automatically provides the binding for `interfaceControllerClassName` (see "The Interface Controller" (page 48), for details).

The WOJavaClientApplet bindings specific to the EODistribution layer are shown in the following table.

Property	Value
useJavaPlugin	If YES, generates HTML that causes Internet Explorer and Netscape browsers to use Sun's Java Plug-in.
distributionContext	The EODistributionContext that the applet uses to handle requests from the client. If no binding is specified, WOJavaClientApplet instantiates one with the session's default editing context and sets the session as the delegate of the distribution context and itself as the invocation target.
interfaceControllerClassName	The name of the initial EOInterfaceController subclass that is created by the Project Builder Assistant when the project is created.
applicationClassName	The name of the EOApplication subclass used for the shared application object.
language	The preferred language for the application. This corresponds to a localized *language*`.lproj` directory in the application's resources. When searching for localized resources, Java Client first looks in this directory, next `English.lproj` (if English is not the preferred language), and finally in the directory for nonlocalized resources.
channelClassName	The class name of the distribution channel to be used by the client, EOHTTPChannel by default.

Other Server Files

A Java Client project includes these other server-side files:

- The Application, Session, and DirectAction class (`.java`) files
- In the Resources group, the model file (`Movies.eomodeld`)
- Also in the Main group, the exported bindings file for the Main component (`Main.api`)

See "Customizing Your Project With Assistants" (page 122) for detailed information on customizing a project.

Creating the User Interface

When you create a Java Client application using the Project Builder Assistant, Project Builder puts a nib file in the Interfaces group of the project. A nib file is primarily a description of a user interface (or part of a user interface). It resides in the Interfaces group of the project, is edited by the Interface Builder application and can be archived along with other resources of your application. This interface file is just like the interface files in typical applications, which are defined using Cocoa objects. The Enterprise Objects palette translates the Cocoa objects into Swing objects that Java Client uses to generate the user-interface on the client.

You typically construct a user interface using Interface Builder, by dragging objects from a palette and dropping them into the content window. Java Client WebObjects applications require that the Enterprise Objects palette be loaded into Interface Builder. This palette includes two objects: EOEditingContext and EODisplayGroup.

Adding the EnterpriseObjects Palette

Before you can create the user-interface for a Java Client application, you have to make sure that Interface Builder has the EnterpriseObjects palette loaded. To do this you need to launch Interface Builder and examine its Preferences window.

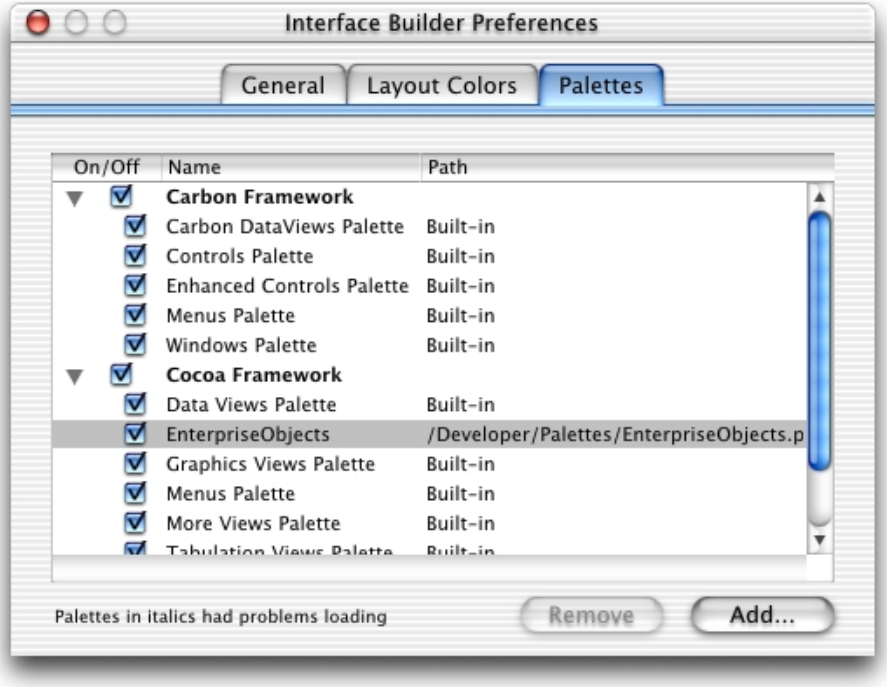
1. Start Interface Builder.

   Navigate to `/Developer/Applications` and launch Interface Builder.

2. Load the required palettes.

   In Interface Builder, choose Interface Builder > Preferences.

   Click the Palettes tab.



   If you don't see the EnterpriseObjects option, you add it by performing these steps:

   a. Click Add.

   b. In the Open Palette dialog, Navigate to the `/Developer/Palettes` directory.

   c. Double-click `EnterpriseObjects.palette`.

Make sure that EnterpriseObjects is selected.

Close the Interface Builder Preferences window.

3. Quit Interface Builder.

   Choose Interface Builder > Quit Interface Builder.

Laying Out the User-Interface Elements

You can construct a basic interface for a Java Client WebObjects application by simply dragging icons from EOModeler into the content window in Interface Builder

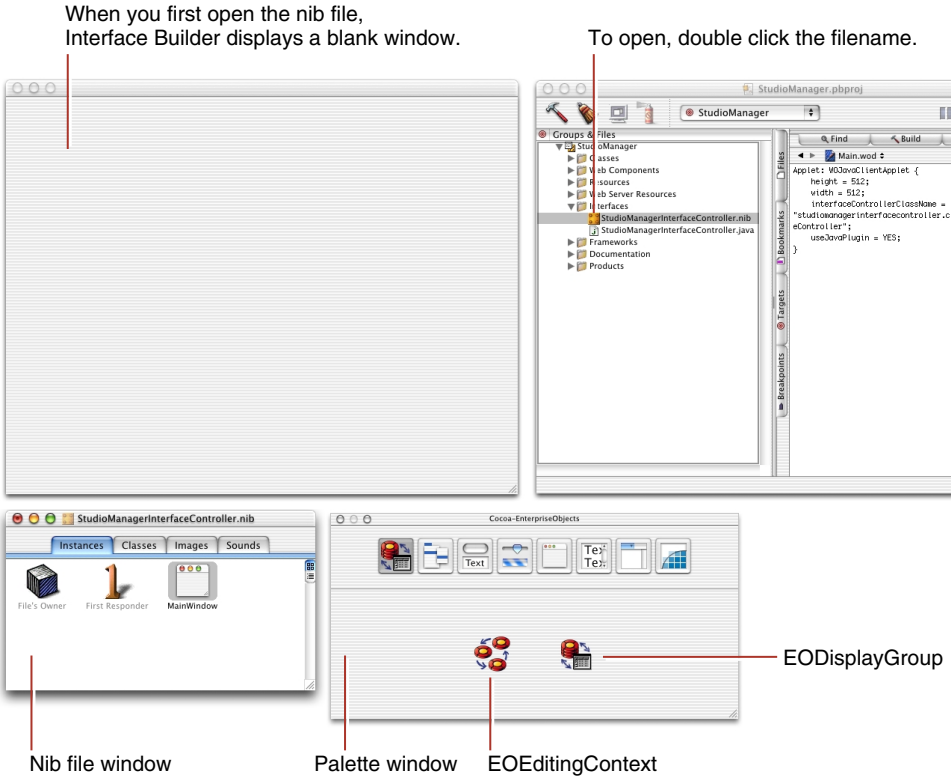
1. Open the StudioManager nib file.

   In the Groups & Files list of the Project Builder main window, open the Interfaces group.

   Double-click `StudioManagerInterfaceController.nib`.

   A blank window (the content window), a nib file window, and a palette window appear when Interface Builder is launched. In Figure 2-3 you can see the windows you'll use to create your application's user interface.

Figure 2-3 The Interface Builder environment



If you don't see the EnterpriseObjects palette, which contains the EOEditingContext and EODisplayGroup elements, you need to add it. For details on how to perform this task, see "Adding the EnterpriseObjects Palette" (page 51).

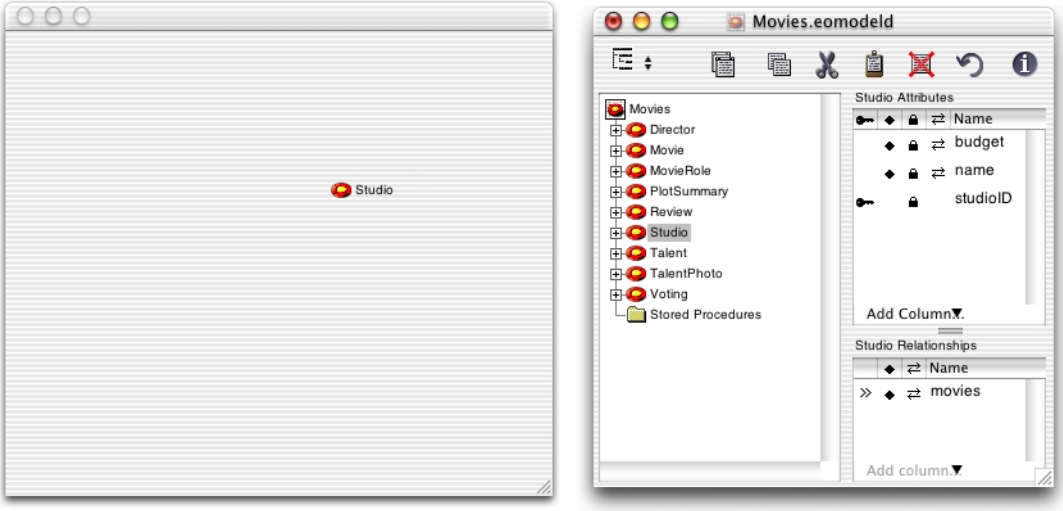
2. Open the model file.

   In the Groups & Files list, open the Resources group.

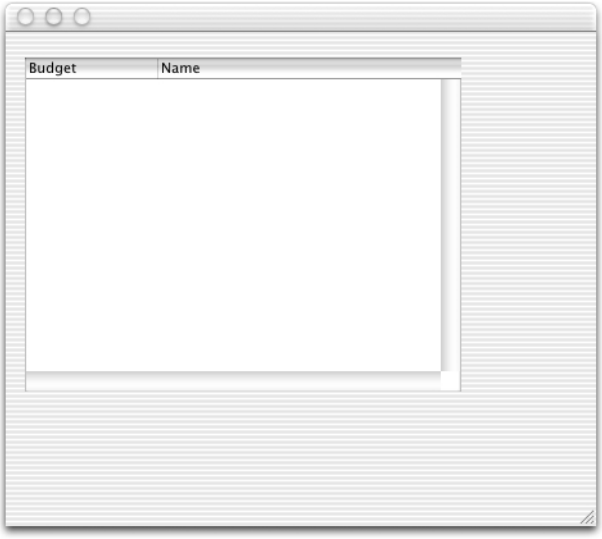
   Double-click `Movies.eomodeld`.

Note: This is a duplicate of the model file you created in "Creating the Movies Model" (page 27). The original model file will not be touched in the remaining of this tutorial.

3. Drag the Studio entity from EOModeler into the content window in Interface Builder.



In the nib file window, there's a new EODisplayGroup named "Studio", 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 a document typically only needs one EOEditingContext, this object is only added once.



(See "What Are EODisplayGroups and EOEditingContexts?" (page 129) for more on display groups and editing contexts.)

An entity EODisplayGroup has keys that correspond to the properties in its associated entity. You can examine these keys in the EODisplayGroup Info window.

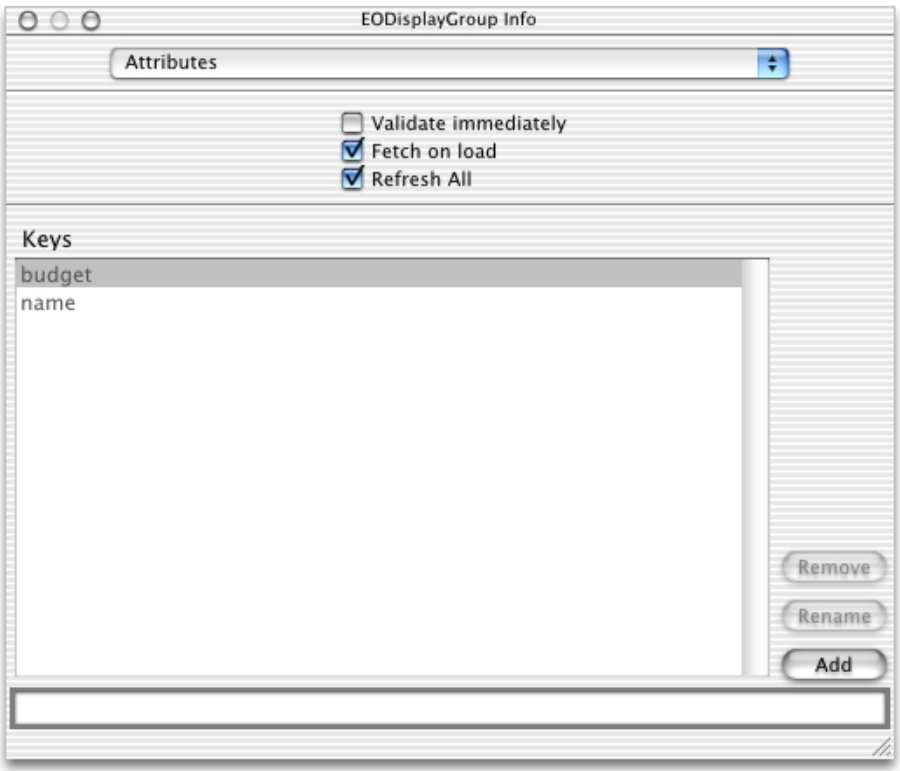
4. Examine the EODisplayGroup in the Info window.

   Select the Studio EODisplayGroup in the nib file window.

Choose Tools > Show Info.

Display the Attributes pane by choosing it from the pop-up menu below the Info window's title bar.

Make sure that "Fetch on load" is selected.



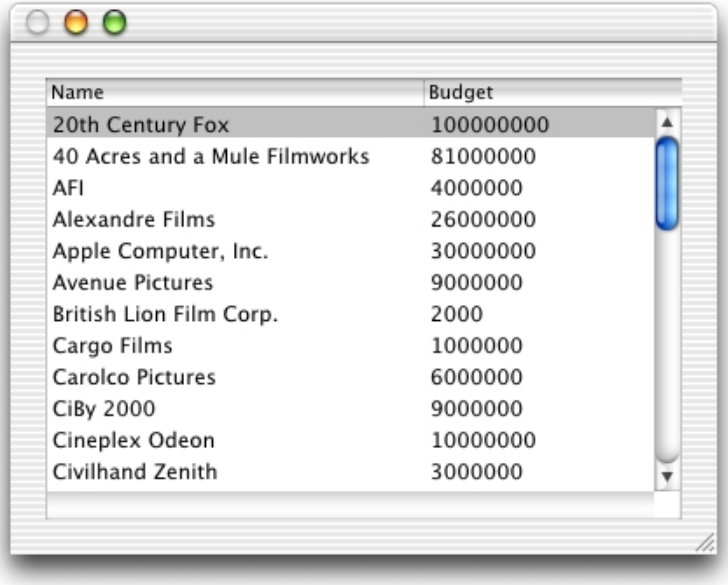
The "Fetch on load" option is important because it allows data to be fetched from the database when you start your application.

The keys listed correspond to the class properties specified for the entity in EOModeler. You can add other keys that are not class properties, such as methods defined in the associated enterprise object class.

The interface that was created when you dragged an entity into the window is already a functional (if simple) application. You can test it.

5. Save the interface.
6. Test the interface.

   Choose File > Test Interface.



Click in the menu bar to end the test.

Note that because the “Fetch on load” option was enabled for the Studio EODisplayGroup in its Info window, the data is automatically fetched when you test your interface.

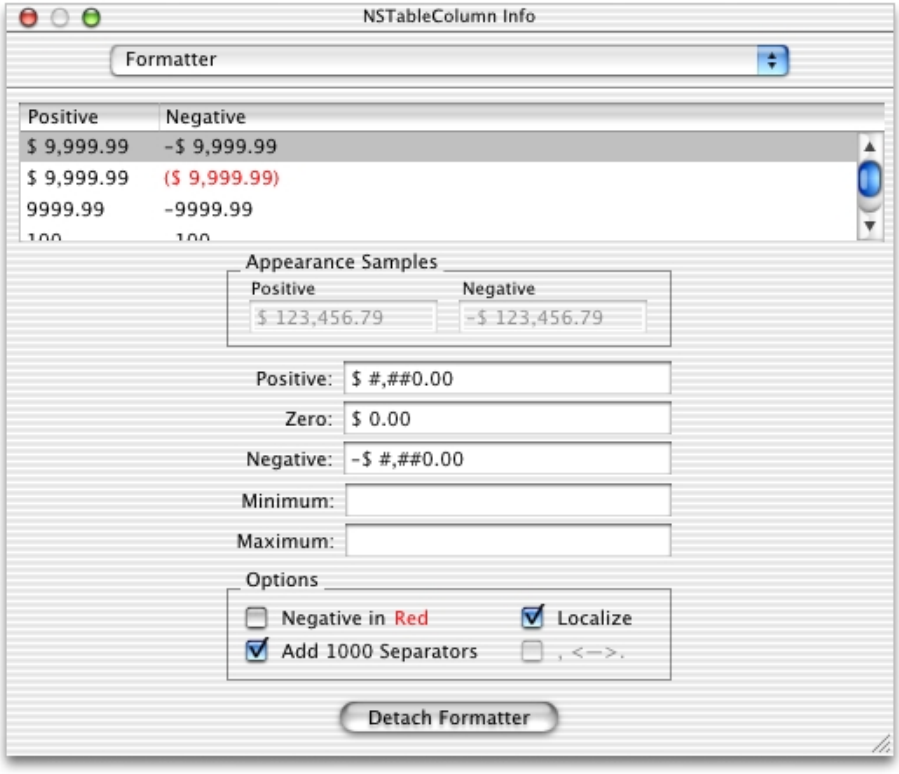
Formatting Currency Values

Notice that the Budget column displays the budget amount for each studio as an unformatted number. You can make the value for the budget attribute display using currency formatting.

To set formatting, follow these steps:

1. Select the Budget column head in the table view. You accomplish this by first double-clicking the table and then clicking the column header.
2. Choose Tools > Show Info.
3. Display the Formatter pane of the Info window.
4. Select a standard currency format.

   Do not set the format to show negative values in red. Colored text is not currently implemented in J2SE.



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, EOEditingContext, and EOInterfaceController 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.

Perform these steps to add action methods to your user-interface:

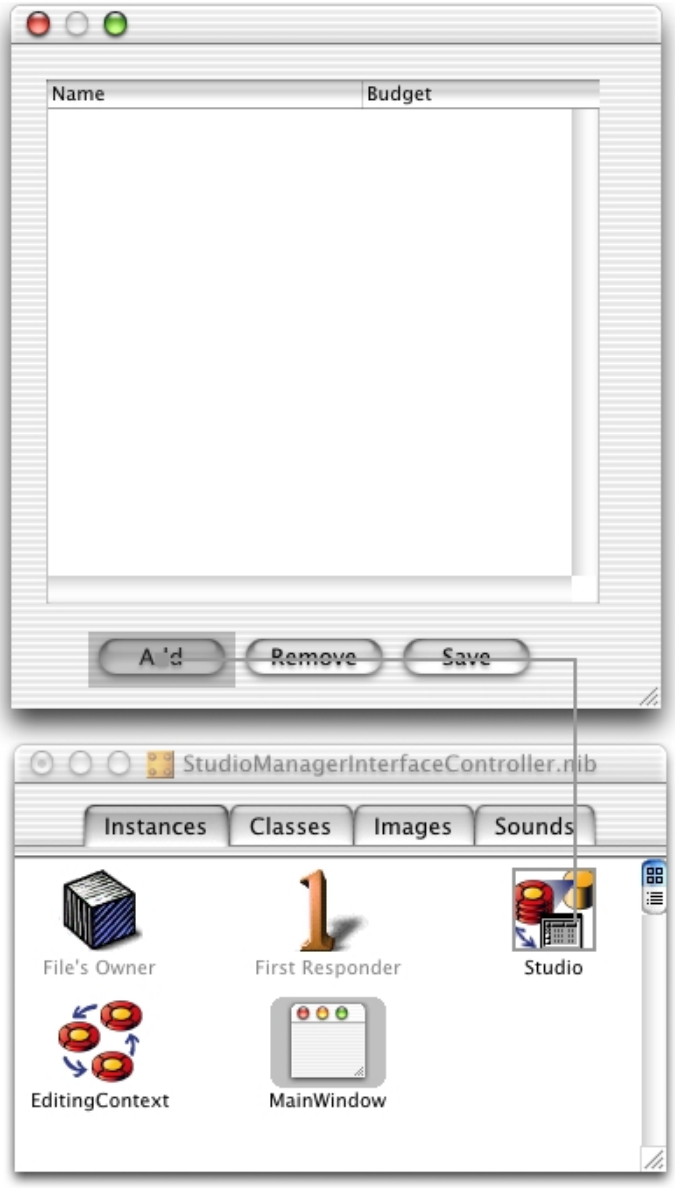
1. Add the interface elements.

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.

2. Connect the Add and Remove buttons to the `insert:` and `deleteSelection:` methods.

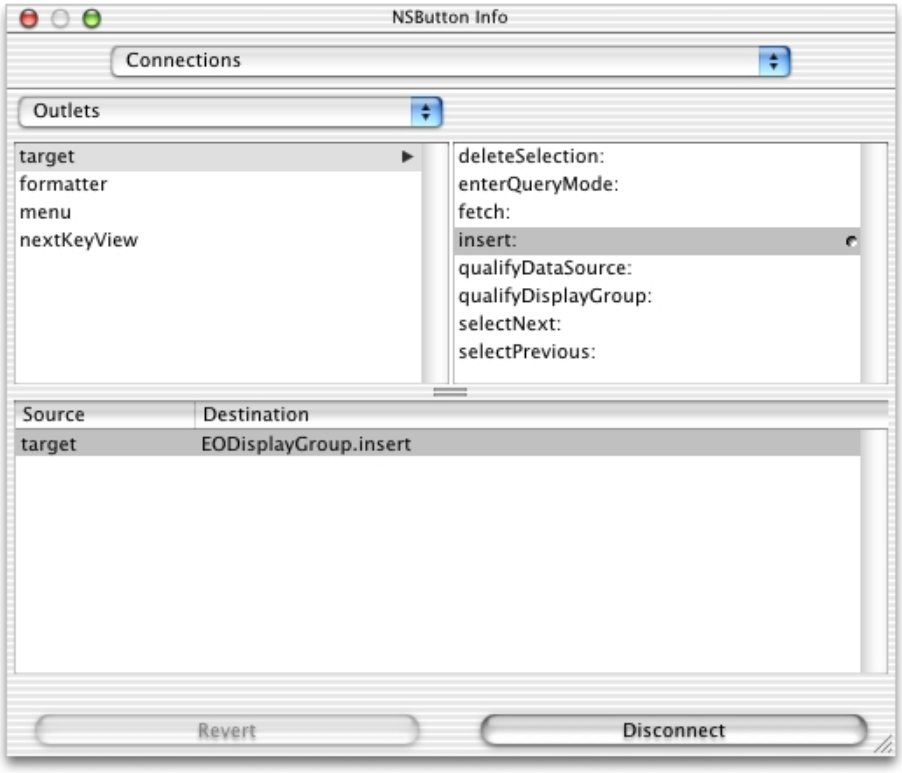
   Control-drag from the Add button to the Studio EODisplayGroup.



In the NSButton Info window, choose Outlets from the pop-up menu at the top of the left column.

Select `target` in the left column.

Select `insert:` in the right column and click Connect.



Using the same process, connect the Remove button to the `deleteSelection:` method.

3. Connect the Save button to the owner's `save()` method

   To connect the Save button, Control-drag from the button to the File's Owner object in the nib file window.

In the Info window of the Save button, choose Outlets in the pop-up menu at the top of the left column.

Select target in the left column.

Double-click `save()` in the right column.

The File's Owner icon represents the object that "owns" the nib file, or the nib file's root object. In a Java Client WebObjects application, this object is an instance of a custom subclass of EOInterfaceController that is automatically created for you (`StudioManager.java`, in this case). EOInterfaceController defines the `save` method and implements it to commit changes to the database.

Note: The EOEditingContext object in the nib file ("EditingContext") also defines a method—`saveChanges`—that commits changes to the database. However, EOInterfaceController's method is preferable because it catches exceptions that might arise from this operation.

Building and Testing Your Application

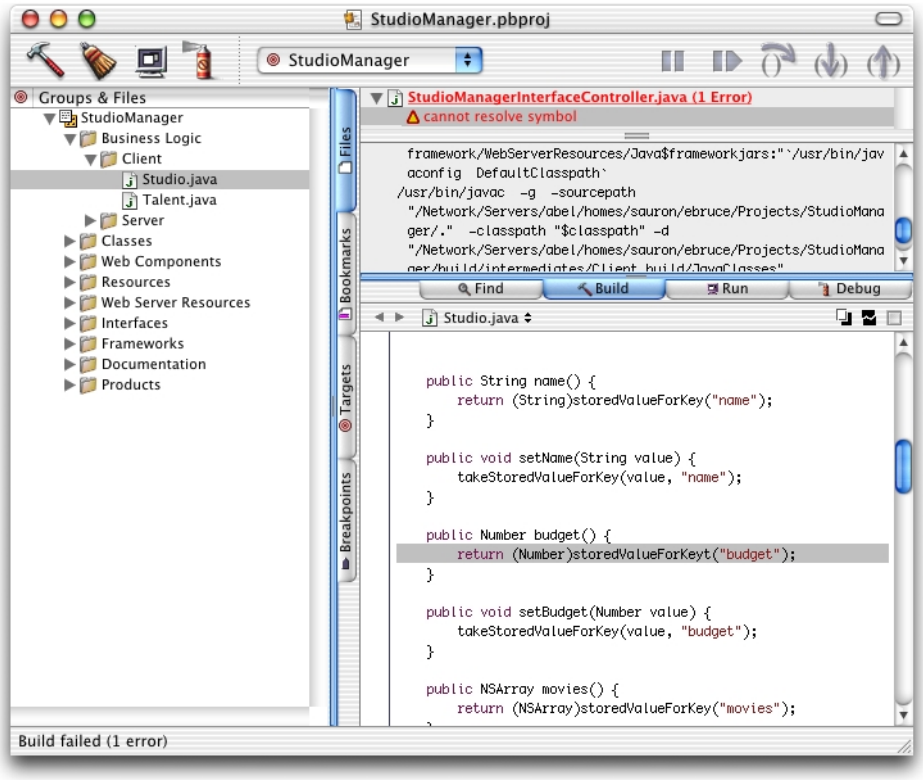
You have now created a Java Client application—a fairly trivial one, to be sure, but still one with all the essential ingredients. Now it is a good idea to build and test your application to catch any problems. Interface Builder gives you a way to test your user interface even before any code is compiled. However, to gauge the complete picture, you still should build your application and test it.

Building the Application

You build a Java Client project using Project Builder.

1. Go to the Project Builder main window and Make sure that the StudioManager target is selected in the target pop-up menu.
2. Click  to build the application.

If there are any coding or linking errors, the Build pane displays them; click an error message in the upper part of the pane to go to the site of the error in the code editor.



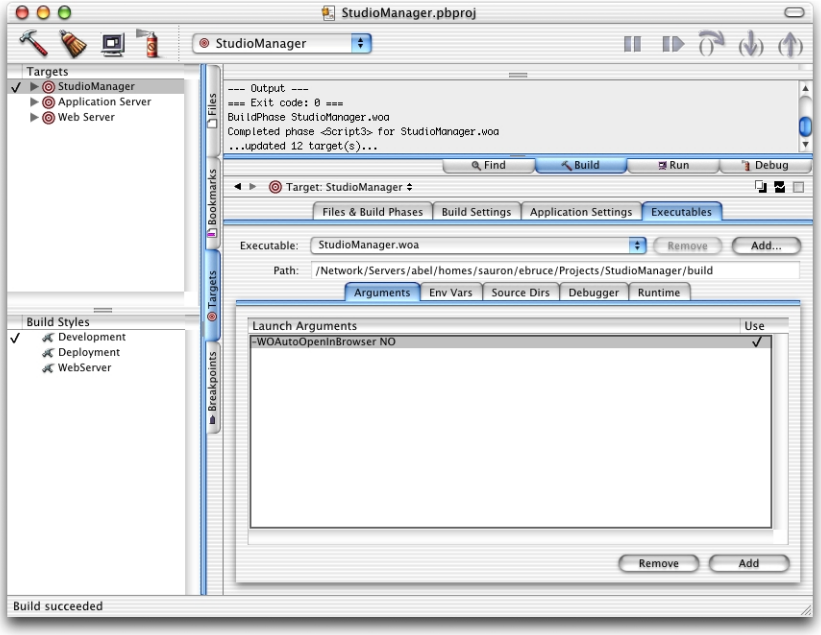
For information about debugging, see "Debugging Java Client WebObjects Applications" (page 121).

Browser Launch by Project Builder

When launching a WebObjects application, Project Builder's default behavior is to launch your Web browser. Unless you are using applets, this is not the desired behavior for Java Client applications. Therefore, you must tell Project Builder not to launch your browser when you run the StudioManager server-side application.

1. Click the Targets tab.
2. Select the StudioManager target in the Targets list.
3. Click the Executables tab in the content area.

4. Click Add on the lower-right corner of the content area.
5. Type `-WOAutoOpenInBrowser NO` in the Launch Arguments column of the Arguments panel.

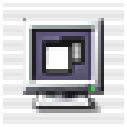


Running a Java Client Application

A Java Client application is really two applications: a server-side application and a client-side application (or applet); they must be running concurrently. You start the server application as you do any WebObjects application in one of the following ways:

- using Project Builder (during development and testing)
- from the command line
- using the Monitor application (the preferred deployment mechanism)

For the procedures for the last two options, check *Serving WebObjects*. You can launch the server application from Project Builder by clicking the Launch icon.



After starting the server application, start the client application; there are several ways to do this:

- **Using the Java interpreter (java):** To start the client as a stand-alone Java application outside a browser, use the `java` interpreter. The syntax for using `java` to start a Java Client application is

  ```
  java [-classpath classpath]
  com.apple.client.eoapplication.EOApplication
  -applicationURLurl
  [-page pageName]
  ```

  You might want to create a script file to make this command automatic and hidden.

  It might not be necessary to specify the `-classpath` option, but if the interpreter cannot find classes, you must either modify your `CLASSPATH` environment variable or add the `-classpath` option to the command. The `-applicationURL` option specifies the application's URL that you would also use in a browser and the `-page` option specifies the name of the page that contains the WOJavaClientApplet component. If `pageName` is not specified, "Main" is assumed.

  Please note that `com.webobjects.eoapplication.EOApplication` is the name of the class that contains the static `main` function that is usually used to start up a Java Client WebObjects application. If you have a different `main` function you must specify the name of the class that implements it instead.

- **Running the client launch script:** This is another way to launch the client as a stand-alone application. The launch script is named after your project, with the `_Client` postfix. It's located in your project folder in `build/StudioManager.woa/Contents/MacOS`. The script takes as its argument the server application's URL (displayed in the console output).

  For example, to run the StudioManager client application, you would type something similar to the following in your shell:

  ```
  cd ~/Projects/StudioManager/build/StudioManager.woa/Contents/MacOS
  ```

```
./StudioManager_Client http://localhost:49387/cgi-bin/WebObjects/
StudioManager
```

- **Using MRJAppBuilder:** You can use the MRJAppBuilder application to build a double-clickable application.
- **Using appletviewer:** The JDK's `appletviewer` tool is very useful during development because it minimizes your start-up time by removing the need to launch a browser. It also lets you view the debugging output inside the shell where you run `appletviewer`. To use the tool, copy the URL of the server application (displayed in the console output) and paste it in a shell window as the argument, for example:

  ```
  appletviewer http://<host>:1234
  ```

  If you are running `appletviewer` on the same machine as the WebObjects application, <*host*> is "localhost"; otherwise it is the host name of the machine on which the application is running.
- **Using browsers:** see explanation below.

There are a few considerations to keep in mind when running a Java Client WebObjects application:

- You can specify `-WOAutoOpenInBrowser NO` on the command line to avoid auto-launching a browser when you start up your server application. See "Browser Launch by Project Builder" (page 65) for more information.
- The `CLASSPATH` environment variable must be correctly set so your application can find all necessary Java classes. If you're using `appletviewer` or the Java interpreter, you need to include the directory containing your client-side classes. The following `CLASSPATH` works for a development environment where the client application is running on the server.

  Type the following command into the Terminal window from which you execute `appletviewer` or the Java interpreter.

  ```
  setenv CLASSPATH "/System/Library/Java:/System/Library/Frameworks
  /JavaVM.framework/Classes/swingall.jar:<DirectoryContainingStudioManager
  >/StudioManager/StudioManager.woa/WebServerResources/Java"
  ```

- `Using Web browsers`: If you run the application in a Microsoft Internet Explorer or Netscape browser, you have to use Sun's Java Plug-in.

These browsers currently do not implement the J2SE specification exactly or have bugs that prevent Java Client applications from working correctly. In particular, Microsoft Internet Explorer does not reset the Java virtual machine, which can cause the application to freeze. In addition, if you start a new applet in a browser that has run another applet, the new applet freezes because the browser's Java virtual machine is not restarted. You will need to restart the browser every time you launch your application, quit the browser and then launch the client. This procedure is not necessary if you use Sun's Java Plug-in. The first time you start an application using the plug-in, the browser will ask you to download the plug-in (the concrete behavior depends on the browser). Afterwards, the plug-in is loaded automatically. Please refer to Sun's documentation at http://java.sun.com/products for more information.

What If It Doesn't Work?

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

- If no data appears in the table, look in the Attributes pane of the Studio EODisplayGroup Info window to make sure that "Fetch on load" is selected.
- If the buttons don't have the desired effect, make sure 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.

What You've Got So Far

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

  Every row in a database is uniquely identified by the value of its primary-key column. 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 currency values and dates
- coordinating the user interface with your data

  Enterprise Objects Framework keeps all parts of an application synchronized 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 is automatically updated to reflect the changes.

Optional Exercise

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

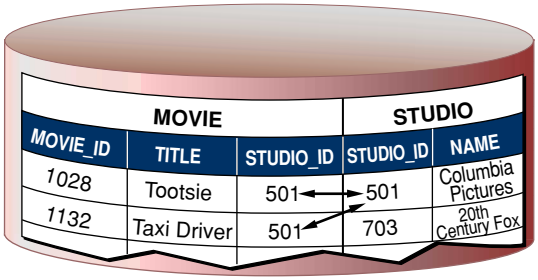
Enhancing the Sample Application

Creating an application that adds and modifies studios is just the first stage of the Studio Manager application. Now you can enhance the application to display all of the movies owned by a selected studio.

Adding Relationships

The Studio, Movie, and Talent entities are not especially interesting when considered separately. Their real significance becomes apparent only 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 has a corresponding Studio. This is represented by both the MOVIE and STUDIO tables having a STUDIO_ID column. In MOVIE, the STUDIO_ID column is a foreign key, while in the STUDIO table 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, your model must specify the corresponding relationships. The example Movies model already defines these relationships. If you used that model in your project, you can skip to "Adding Movies to the Application" (page 75). Otherwise, you'll need to add relationships using EOModeler.

Using EOModeler to Add Relationships

To add the relationships between Movie and Studio in EOModeler, follow these steps:

1. Display the graphical view of the model.

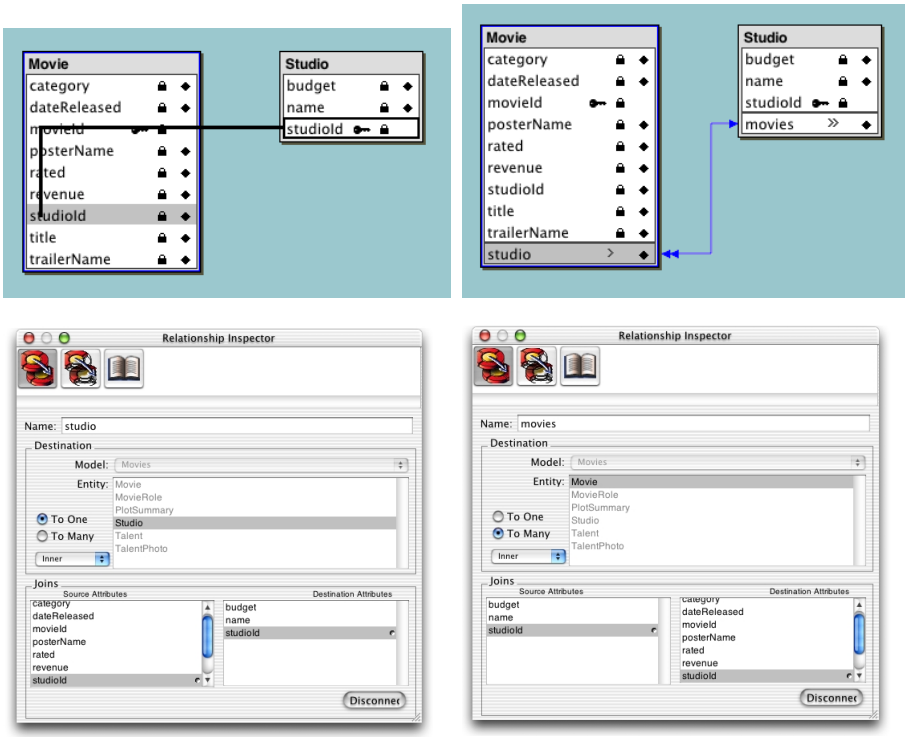
   Choose Tools > Diagram View.

   You are presented with a graphical view of the entities in your model. Because no relationships have been defined, you'll see only a group of individual entities.

2. Create the Movie-Studio relationships.

   Position the Movie and Studio entities next to each other.

   Control-drag from the studioId attribute of the Movie entity to the studioId attribute of the Studio entity.



You'll add more relationships to the model in "Expanding the Movies Model" (page 87), using a different technique.

3. Add client-side class properties.

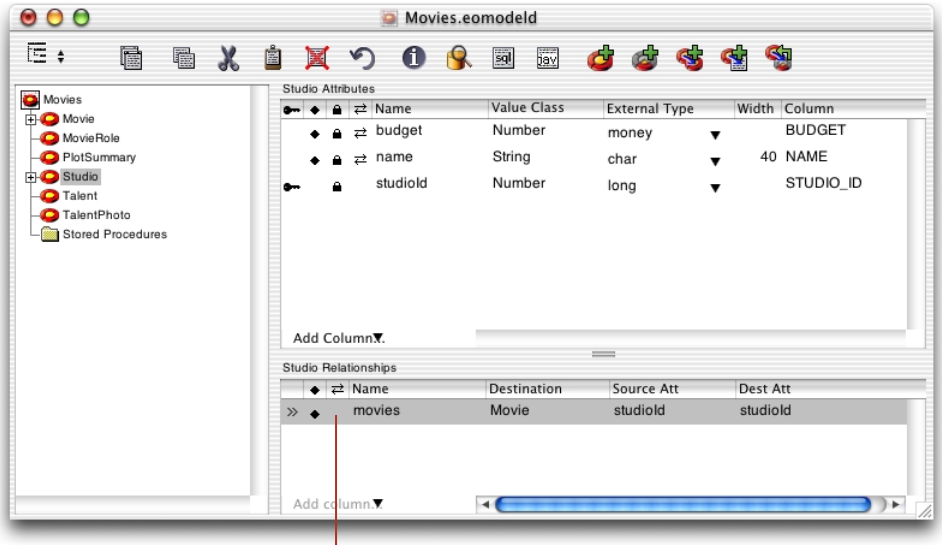
   For the client to be able to access the objects referenced by the relationships of each entity, they have to be defined as client-side class properties. EOModeler does not automatically select this option for new relationships.

   Select Tools > Table Mode.

   Select the Studio entity.

   In the Studio Relationships table, click the ⇄ column in the movies relationship information row. When the arrows show up, it means that the relationship is a client-side class property.

   Repeat the process for the studio relationship of the Movie entity.



4. Save the model.

You've just defined two relationships in the Movies model:

- a to-one relationship called "studio" in the Movie entity that references each Movie object with the Studio object that owns it
- a to-many relationship called "movies" in the Studio entity that references all the Movie objects that belong to a particular Studio object

As you can see, there are two types of relationships:

- **to-one** relationships associate a *source* object to a single *target* object. These relationships are named using the singular form of the target entity name.
- **to-many** relationships associate a source object to one or more target objects. These relationships are named using the plural form of the target entity name.

Adding Movies to the Application

The relationships defined in the Movies model now come into play in your application. You can use Studio's movies 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 holds 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. In the Studio Manager application, Studio is the master table and Movie is the detail table.

Creating a Master-Detail Interface

Starting with this exercise, you will create the final user interface of the Studio Manager application. This means that you should start off by removing all objects added earlier to your nib file; also, give your application window a title.

1. Prepare the nib file.

   In Interface Builder, delete the table view from the window.

   Delete the Studio EODisplayGroup and the EditingContext from the nib file window.

   Save the nib file.

2. Set the window title.

   Select the window by clicking its title bar.

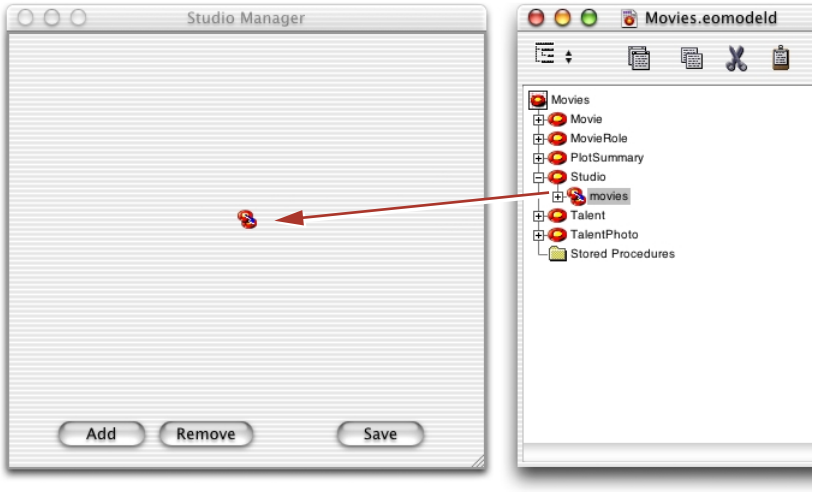
   Choose Show Info from the Tools menu.

   Enter `Studio Manager` in the Title field of the Attributes pane and press Enter.

3. Create a master-detail interface.

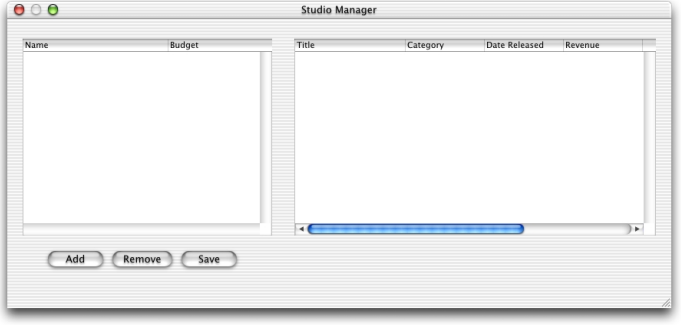
   You can create a master-detail interface by simply dragging a relationship from EOModeler onto your window.

   Drag the Studio entity's movies relationship from EOModeler onto the window in Interface Builder.



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 displayed to the user.

Rearrange and resize the tables so that they are side by side.

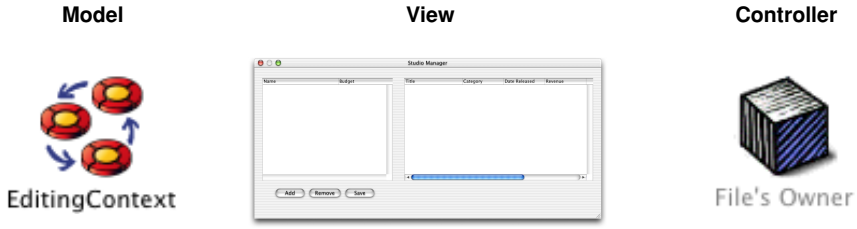


Reconnect the Add and Remove buttons to the new Studio EODisplayGroup.

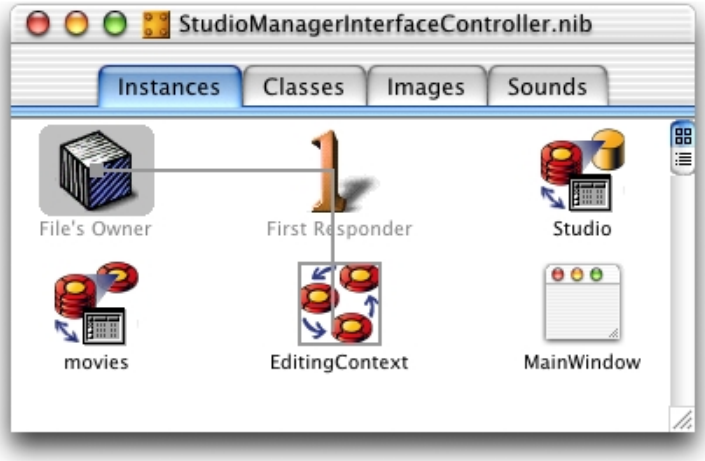
(See "Adding Action Methods" (page 60) for these procedures.)

4. Connect the interface controller to its editing context and display group.

For a Java Client application, the interface controller—represented by the File's Owner icon in the nib file window—is the *controller* object in the Model-View-Controller design scheme. The interface controller comes already connected to its *view* through the component outlet. But you must connect it to its *model* object, the editing context.

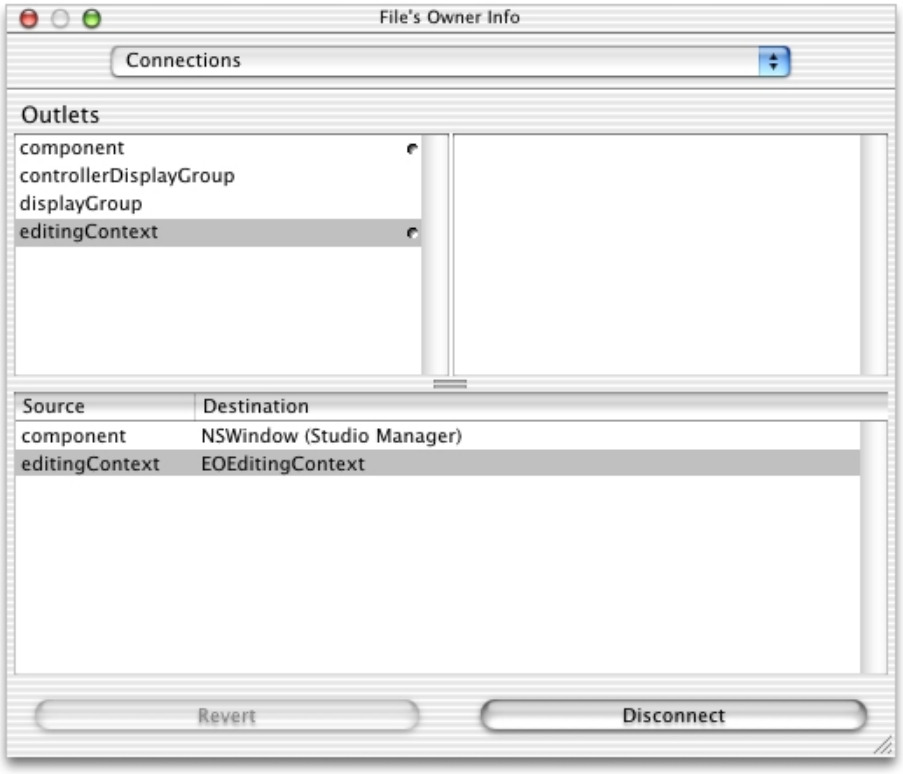


Control-drag from the File's Owner icon to the EditingContext icon in the nib file window.



In the Connections pane of the Info window, select the `editingContext` outlet.

Click Connect.



Control-drag from the File's Owner icon to the Studio icon in the nib file window.

In the Connections pane of the Info Window, select the `displayGroup` outlet.

Click Connect. (Alternatively, you can double-click the `displayGroup` outlet to connect it to the File's Owner.)

5. Add titles to the Studio and Movies tables.

   Drag a Message Text element from the palette window into the content window, above the Studio table.

   Choose Tools > Show Info.

   Display the Attributes pane of the Info window.

   Enter `Studio` in the Title field.

Repeat the process for the Movies table.

6. Change the formatting for currency and dates.

   Select the Budget column in the Studio table.

   Choose Tools > Show Info.

   Display the Formatter pane of the Info Window.

   Select the standard currency formatter (the one that does not display negative numbers in red).

   Repeat the process for the Revenue column in the Movies table.

   Select the Date Released column in the Movies table.

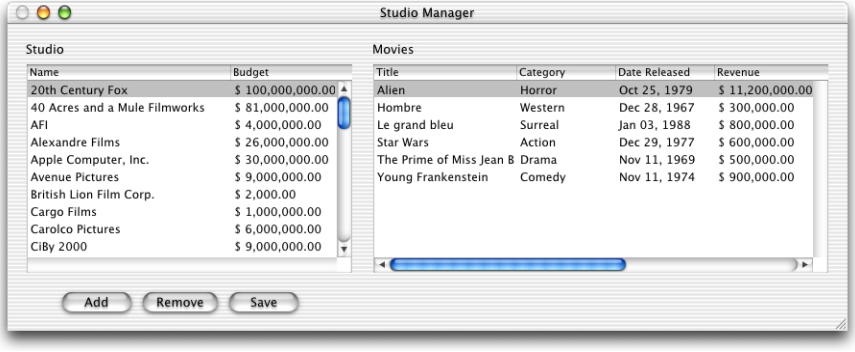
   Display the Formatter view of the NSTableColumn Info window.

   Select the formatter that uses the "%b %d, %Y" format.

7. Save the interface file.

8. Test the interface.

   Choose File > Test Interface.



Choose Interface Builder > Quit Interface Builder to end the test.

9. Learn the size of the window.

   To see the effects of your changes, you must compile and run the application. However, before you run the application, there is one last step to perform. The applet providing the running environment for your Java Client application is set to a default size in the WOJavaClientApplet bindings in `Main.wod`. This size could be too small to accommodate your user interface (or too large for it).

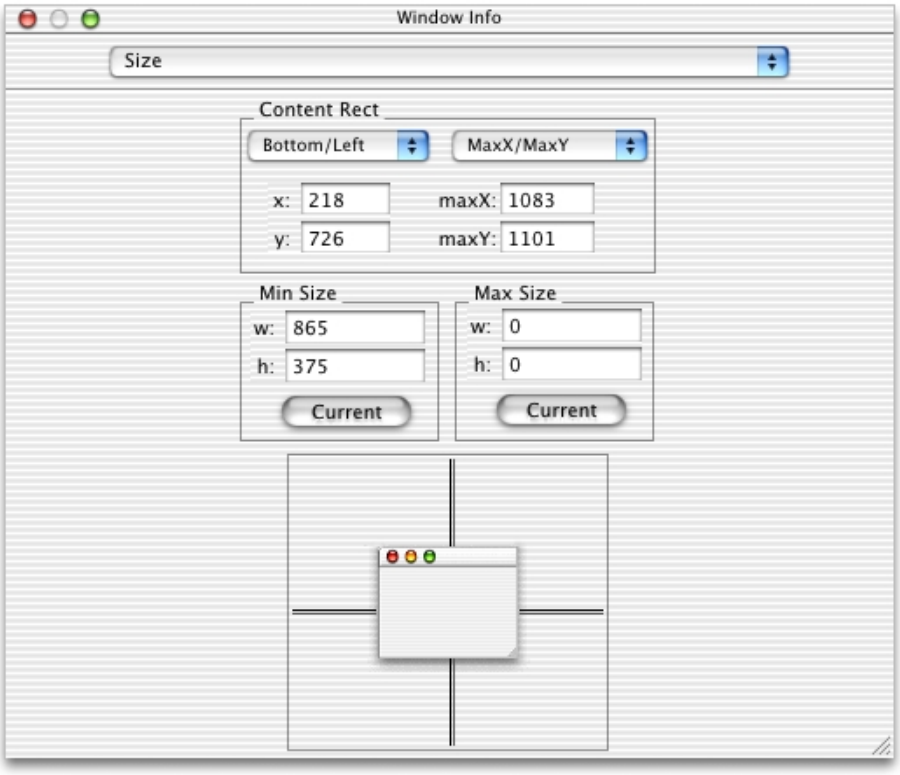
In Interface Builder, select the content window by clicking its title bar.

Choose Tools > Show Info.

Display the Size pane.

Click Current in the Min Size group.

Write down the values for the `w` (width) and `h` (height) attributes.



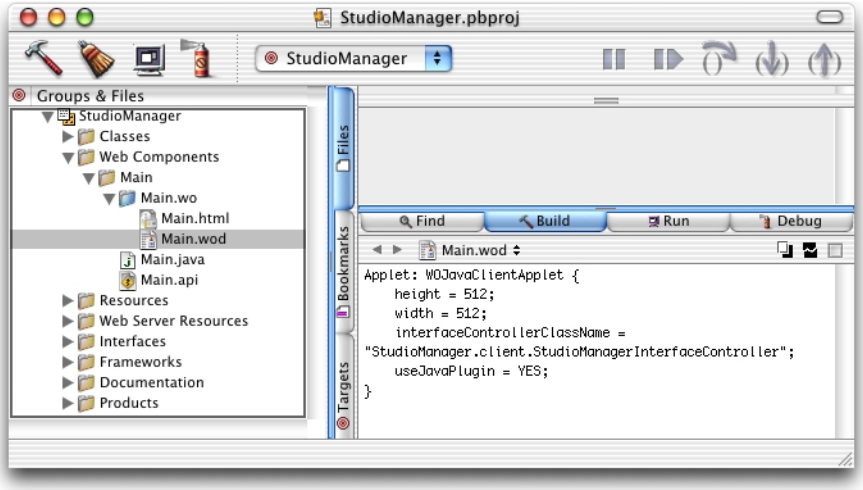
10. Enter the size information into the Main component's WOJavaClientApplet's subcomponent's size bindings.

    In the Groups & Files list of Project Builder's main window, click the disclosure triangle to the left of the Web Components group.

    In the Web Components group, click the disclosure triangle to the left of the Main subgroup.

In the Main subgroup, click the disclosure triangle to the left of the Main.wo subgroup.

In the Main.wo subgroup, select `Main.wod`.



In Project Builder, select Web Components > Main > Main.wo > `Main.wod`.

Enter the dimensions of the window in the height and width bindings.

```
Applet: WOJavaClientApplet {
    height = 375; // change this
    width = 865; // and this
    interfaceControllerClassName =
“studiomanagerinterfacecontroller.client.StudioManagerInterfaceControlle
r”;
    useJavaPlugin = YES;
}
```

Save `Main.wod`.

Now you are ready to test the application.

11. Build the application.

    See “Building the Application” (page 64) for details.

12. Run and test the application.

Click  to launch the server application.

Launch the client application.

See "Running a Java Client Application" (page 66) for details on launching Java Client applications.

When you select a studio in the Studio table, the display changes in the Movies table to show the selected studio's movies.

Transferring Movies Between Studios

One of the primary functions of the Studio Manager 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 Movies table view.

1. Add a pop-up list.

   Drag a pop-up list (labeled "Item1" on the Views palette) into the window.

   Put the pop-up list directly below the Studio table view and leave some space between it and the row of buttons. Later you will be adding fields between the pop-up list and the buttons. See Figure 3-1 (page 86) for a guide.

2. Set the binding that provides the pop-up list with the list of studio names.

   The pop-up list needs a source from which to obtain the list of options to display when the user clicks on it, in this case, a list of studio names. You need to connect the pop-up lists `titles` aspect with the Studio EODisplayGroup's `name` property.

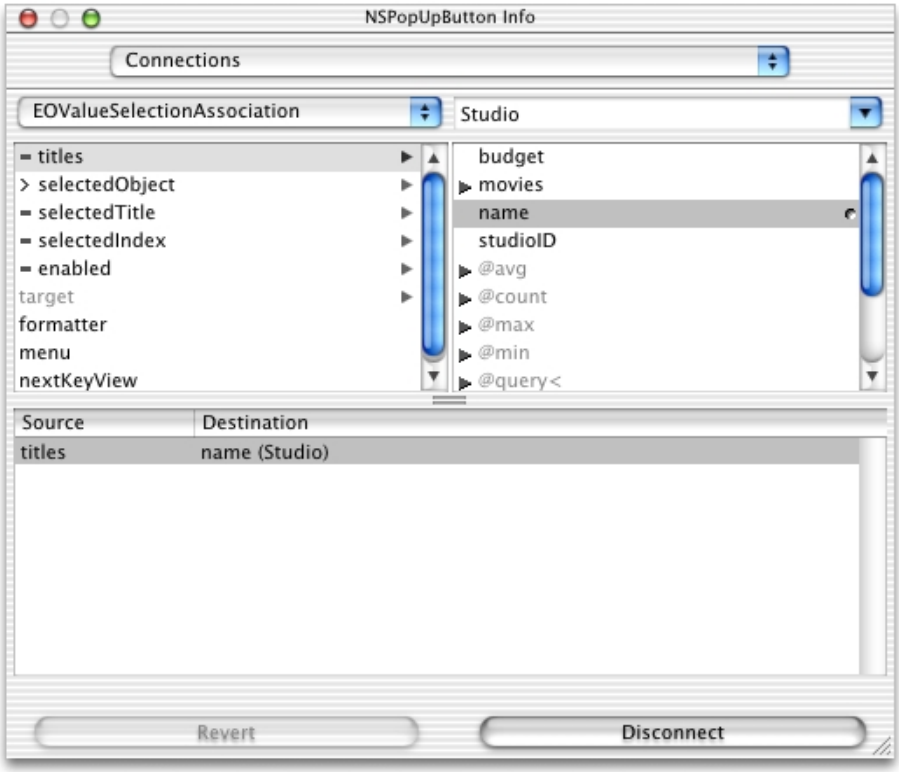
   Control-drag from the pop-up list to the Studio EODisplayGroup.

   In the NSPopupButton Info window, select EOValueSelectionAssociation 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 list.

   Select `name` in the right column (since you want to display studio names in the pop-up list).

Click Connect.



3. Set the binding that changes the studio of the selected movie.

   Now you have to add another binding to the EOValueSelectionAssociation so that when you change the selected studio title, it sets the corresponding `studio` relationship property in the selected Movie object. In other works, when you choose a studio from the pop-up list, the value of the `studio` property of the selected movie is set to the studio selected in the pop-up list.

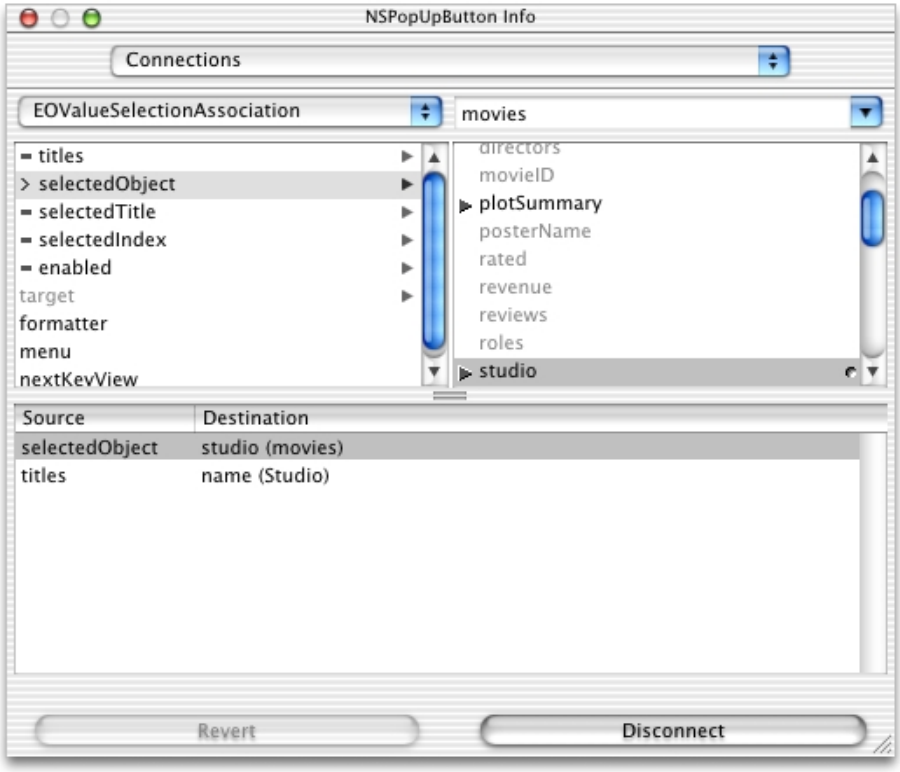
   Control-drag from the pop-up list to the movies EODisplayGroup.

   Display the EOValueSelectionAssociation pane of the Info window.

   Select `selectedObject` in the left column.

   Select `studio` in the right column.

Click Connect.

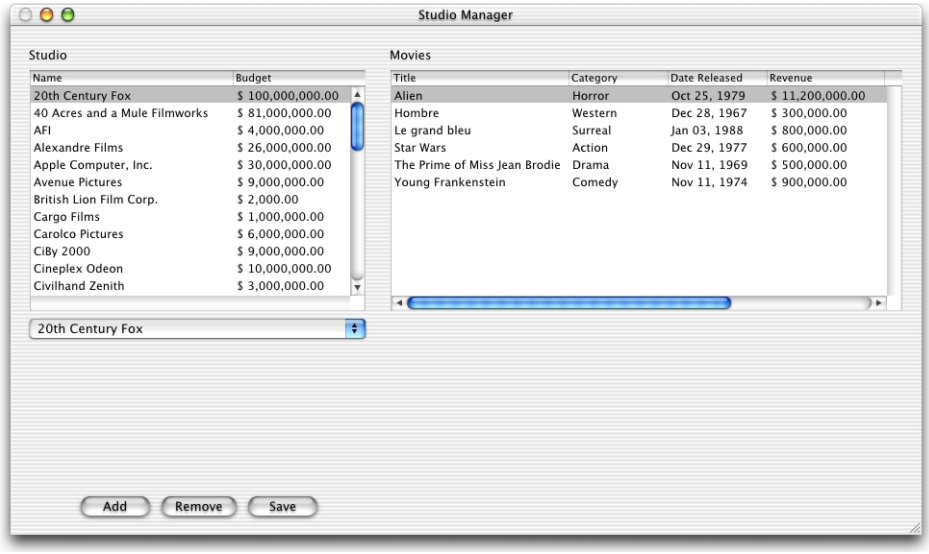


The `selectedObject` aspect is bound to the relationship property (in this example, Movie's `studio` property) that corresponds to the object bound to the `titles` aspect (Studio).

4. Save the interface.
5. Build and test-run the application.

(See "Building and Testing Your Application" (page 64) for details.)

Figure 3-1 Master-detail interface



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.

For more information on associations, see "What Is an Association?" (page 130).

Expanding the Movies Model

You are almost ready to add custom behavior to your enterprise objects. But first you need to add additional relationships to the Movies model.

If you used the example Movies model, you can skip most of this section. However, you should, review Table 3-1 (page 89) to make sure that the relationship names in the your model match the ones defined in the table.

Movies tell stories using a series of events or *plot*. In the plot, people or objects interact according to their particular roles in the movie. Those *movie roles* are portrayed by actors, or *talent*. To help cast a movie, actors' *photos* can be examined to determine if they have the appropriate "look" for a role.

In the Movies model, Movie is associated to PlotSummary through the `plotSummary` relationship. There are reciprocal relationships between the Movie and MovieRole objects. The Movie entity has a `movieRoles` relationship that associates a Movie with its MovieRoles. In turn, the MovieRole entity has a `movie` relationship that associates a MovieRole with the Movie that it belongs to.

The Talent entity has relationships that associate it with the TalentPhoto and MovieRole entities. The `movieRoles` relationship determines the roles the Talent object (actor) stars in. The `photo` relationship associates Talent objects with the actor's picture, or TalentPhoto object.

You will now add the remaining relationships to the model.

1. Open the Movies model file.

   In the Groups & Files list of Project Builder's main window, open the Resources group.

   Double-click `Movies.eomodeld`.

2. Add the Movie to MovieRole relationship.

   a. Create the relationship.

      Select the Movie entity.

      Choose Property > Add Relationship.

Choose Tools > Inspector.

Select To Many in the Destination group.

Select MovieRole as the destination entity.

Select movieId as the source attribute in the Joins group.

Select movieId as the destination attribute.

Click Connect.

EOModeler names the relationship `movieRoles` because the relationship's target is MovieRole and it's a to-many relationship. It is strongly recommended that you use the names that EOModeler provides as they describe both the target of the relationship and its type. However, you are free to use a naming convention that will help the users of your model to easily understand it.

b. Make the relationship a client-side class property.

In the Movie Relationships Table, click in the ⇄ column in the movieRoles relationship information row, to make it a client-side class property.

3. Add the MovieRole to Movie relationship.

a. Create the relationship.

Select the MovieRole entity.

Choose Property > Add Relationship.

Ensure To One is selected in the Destination group.

Select Movie as the destination entity.

Ensure movieId is selected as the source and destination attribute.

Click Connect.

EOModeler names the relationship "movie."

b. Make the relationship a client-side class property.

In the MovieRoles Relationships table, click in the ⇄ column in the movie relationship information row, to make it a client-side class property.

4. Add the remaining relationships.

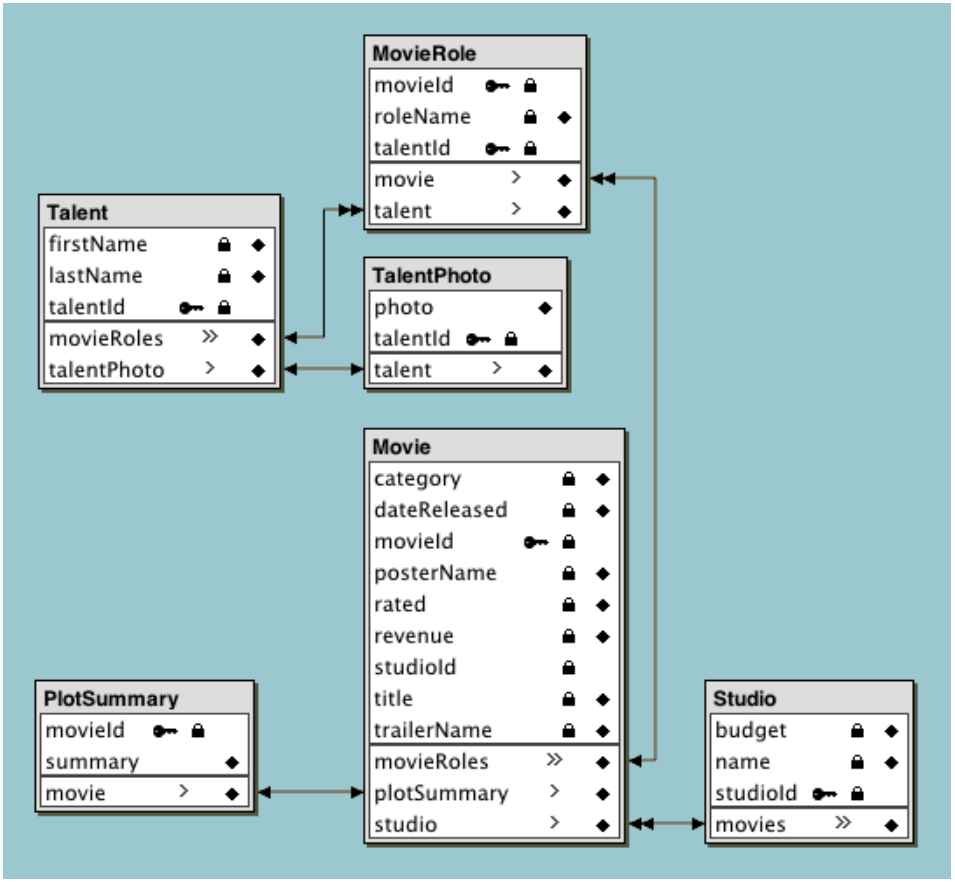
Table 3-1 lists all the relationships that the Movies model should contain. Make sure all of them are entered in your model. Also remember to make them client-side class properties (see "Using EOModeler to Add Relationships" (page 72) for details).

Table 3-1 Movies model relationships

Source	Destination	Name	Type	Attribute
Movie	MovieRole	`movieRoles`	to-many	`movieId`
Movie	PlotSummary	`plotSummary`	to-one	`movieId`
Movie	Studio	`studio`	to-one	`studioId`
MovieRole	Movie	`movie`	to-one	`movieId`
MovieRole	Talent	`talent`	to-one	`talentId`
PlotSummary	Movie	`movie`	to-one	`movieId`
Studio	Movie	`movies`	to-many	`studioId`
Talent	MovieRole	`movieRoles`	to-many	`talentId`
Talent	TalentPhoto	`talentPhoto`	to-one	`talentId`
TalentPhoto	Talent	`talent`	to-one	`talentId`

When finished, your model's diagram should look like the one in Figure 3-2.

Figure 3-2 Diagram of the Movies model



At this point your model is complete.

Adding Behavior to Your Enterprise Objects

As the preceding sections illustrate, you can go quite far in a Java Client application without writing any code. However, the real power of such an application or any 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

When creating a model of a database using the EOModeler application's wizard, you have the option of creating custom enterprise objects. If this option is selected, EOModeler derives both entity name and class name from the name of the associated database table. Otherwise, EOModeler maps entities to the EOGenericRecord class, which can be thought of as the default enterprise object class.

The EOGenericRecord class is sufficient when all you want the entity to do is get and set properties. 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.

To use a custom class instead of EOGenericRecord follow these steps:

1. In EOModeler, select the Movies model root (top of the tree).

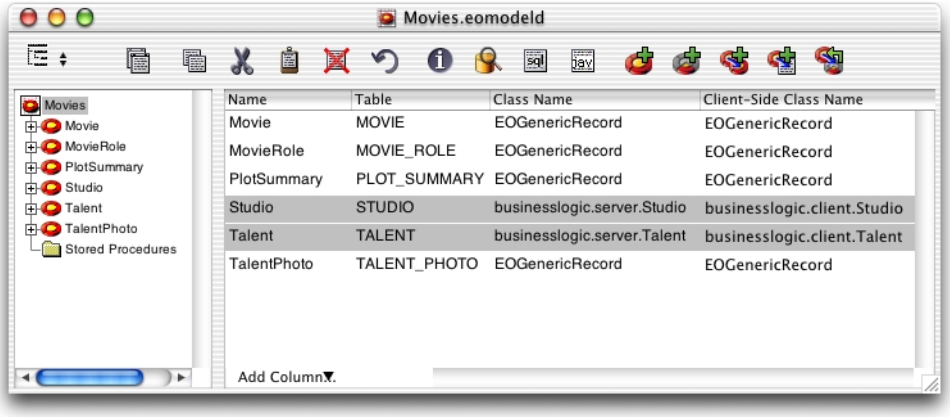
   Make sure you're in Table mode.

   If the Client-Side Class Name column is not visible, choose Client-Side Class Name from the Add Column pop-up menu at the bottom of the window.
2. Double-click the Studio's Class Name cell in the table.
3. Type `businesslogic.server.Studio` in the cell (`businesslogic.server` is the package name).
4. Double-click the Client-Side Class Name cell.

5. Type `businesslogic.client.Studio` in this cell (`businesslogic.client` is the package name).

Repeat the above steps for the Talent entity (append "Talent" to the package names).

Save the model.



By convention, the names of classes (minus the package prefix) are based on the name of the corresponding entity and the initial letter of the name is capitalized.

There is no requirement that you create matching server and client classes. You can implement a class only on the server or the client, whichever suits your needs; the unimplemented class assumes the default behavior of EOGenericRecord.

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

For more information on custom enterprise objects, see "When Do You Use a Custom Enterprise Object Class?" (page 131).

Getting Your Project Ready to Receive Custom Classes

Project Builder stores most of the files for a project at the top level of the project directory. To separate *business logic* files from normal WebObjects files, it is recommended that a group called Business Logic be created in the project. In it, all the business logic files can be stored. To further separate server-side files from client-side files, the Business Logic group should contain at least two subgroups: Server and Client. And additional subgroup, Common, can be added under Business Logic when you want to share behavior with the server and client applications.

To create the recommended grouping for Java Client projects, follow these steps:

1. Create the directory structure.

   Create the following directories at the top level of your project directory:

   `BusinessLogic`

   `BusinessLogic/Server`

   `BusinessLogic/Client`

   `BusinessLogic/Common` (optional, not needed for this tutorial)

2. Create the Business Logic group in the project.

   In the Groups & Files list of Project Builder's main window, select the StudioManager root group.

   Choose Project > New Group.

   Name the group "Business Logic".

Generating Source Files

To begin creating your custom classes, generate source files for the Studio and Talent entities. You'll use these source files as a basis for adding custom behavior to your enterprise objects. Generating source files in a Java Client application typically produces skeletal Java (`.java`) files for the associated class. You then add these files to your project.

Note: To generate source files for an entity, you must have replaced the text "EOGenericRecord" in the Class Name and Client-Side Class Name fields with a package name concatenated with a class name.

1. Generate the client-side `.java` files.

   Open the Movies model file (if it's not already open).

   Select the Studio entity.

   Choose Property > Generate Client Java Files.

   Select the Client directory inside the BusinessLogic directory.

   Click Save.

Repeat the process for the Talent entity.

2. Generate server-side `.java` files.

   Select the Studio entity.

   Choose Property > Generate Java Files.

   Select the `Server` directory inside the `BusinessLogic` directory.

   Click Save.

   Repeat the process for the Talent entity.

When EOModeler generates a class file (such as `Studio.java`), it strips off the package prefix and inserts a package declaration near the top of the file. The class file also includes the necessary import declarations as well as the instance variables and accessor methods derived from the properties of the entity as defined in the model file.

Adding Custom Java Files to Your Project

After you have generated custom Java files for the enterprise objects that you wish to customize, you can add them to your project. Remember that you must first prepare your project file as explained in "Getting Your Project Ready to Receive Custom Classes" (page 93).

1. Add the custom client-side classes to the project file in Project Builder.

   In the Groups & Files list in Project Builder's main window, select the Business Logic group.

   Choose Project > Add Files.

   Select the `Client` directory in the `BusinessLogic` directory.

   Click Open.

   The target selection sheet appears.

Select the Client target.

Click Add.

2. Add the custom server-side classes.

   In the Groups & Files list in Project Builder's main window, select the Business Logic group.

   Choose Project > Add Files.

   Select the `Server` directory in the `BusinessLogic` directory.

   Click Open.

   The target selection sheet appears.

   Select the Server target.

   Click Add.

Now the project uses custom classes for the Studio and Talent enterprise objects instead of EOGenericRecord. These class files can now be edited to implement custom behavior.

If you examine the code in Listing 3-1, you'll notice that the class generated by EOModeler does not have actual instance variables or *fields*. The methods to access the attributes of the custom enterprise object are implemented using key-value coding.

Listing 3-1 Client-side Studio.java file generated by EOModeler

```
package businesslogic.client;

import com.webobjects.foundation.*;
import com.webobjects.eocontrol.*;

public class Studio extends EOGenericRecord {

    public Studio() {
        super();
    }

    public Studio(EOEditingContext context, EOClassDescription classDesc,
EOGlobalID gid) {
        super(context, classDesc, gid);
    }


    public String name() {
        return (String)storedValueForKey("name");
    }

    public void setName(String value) {
        takeStoredValueForKey(value, "name");
    }

    public Number budget() {
        return (Number)storedValueForKey("budget");
    }

    public void setBudget(Number value) {
        takeStoredValueForKey(value, "budget");
    }
```

```
    public NSArray movies() {
        return (NSArray)storedValueForKey("movies");
    }

    public void setMovies(NSMutableArray value) {
        takeStoredValueForKey(value, "movies");
    }

    public void addToMovies(EOGenericRecord object) {
        NSMutableArray array = (NSMutableArray)movies();

        willChange();
        array.addObject(object);
    }

    public void removeFromMovies(EOGenericRecord object) {
        NSMutableArray array = (NSMutableArray)movies();

        willChange();
        array.removeObject(object);
    }
}
```

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.

For more information, see "Adding Behavior to Enterprise Objects" (page 131).

Distributing Business Logic in Java Client Applications

The value of Java Client applications, of course, lies in their ability to distribute processing duties among objects on the server and objects on the client. Primarily for security and performance reasons, you can have only objects on the server performing some tasks and only objects on the client performing others.

For example, sometimes you want only objects behind the firewalls and other security mechanisms of the server to have access to sensitive information, such as account numbers. On the other hand, processing tasks such as calculation of balances should be performed by objects on the client, thereby improving application performance by eliminating the need for a cycle of the request-response loop.

There are no hard and fast rules for how to distribute object behavior. An enterprise object on the client can have the same set of methods and instance variables as its counterpart on the server, or what it has can be a subset (or superset) of the other object's methods and instance variables. The best way to distribute business logic among objects depends on the particular nature of your application.

Managing Relationships

In "Transferring Movies Between Studios" (page 83) 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` when you add an object to an array that represents a relationship property, and invokes `removeObject` when you delete an object from the array. These methods are part of the EORelatioshipManipulation protocol. See the *EOControl Java API Reference* for details.

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 the source code you created using EOModeler in "Generating Source Files" (page 94).

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 of the Client target, as shown in Listing 3-2.

Listing 3-2 Studio.java (server and client) - Extending default behavior

```
import java.math.*;

public void addToMovies(EOGenericRecord object) {
        NSMutableArray array = (NSMutableArray)movies();

        willChange();

        // Subtract movie's revenue from budget.
        Number newBudget;
        Number movieRevenue = (Number)object.storedValueForKey("revenue");
        newBudget = new BigDecimal(budget().doubleValue() -
movieRevenue.doubleValue());
        setBudget(newBudget);

        array.addObject(object);
    }

    public void removeFromMovies(EOGenericRecord object) {
        NSMutableArray array = (NSMutableArray)movies();

        willChange();

        // Add movie's revenue to budget.
        Number newBudget;
        Number movieRevenue = (Number)object.storedValueForKey("revenue");
        newBudget = new BigDecimal(budget().doubleValue() +
movieRevenue.doubleValue());
        setBudget(newBudget);

        array.removeObject(object);
    }
```

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 times 1.5 constitutes the studio's portfolio value. To calculate a studio's portfolio value, you could have a method in `Studio.java` like the one shown in Listing 3-3.

Listing 3-3 Studio.java (client) - Calculating a studio's revenue

```
import java.math.*;
public Number portfolioValue() {
    int i, count;
    double total;
    NSArray revenues;

    total = 0.0;
    revenues = (NSArray)(movies().valueForKey("revenue"));

    count = revenues.count();
    for (i = 0; i < count; i++) {
        total +=
            ((Number)(revenues.objectAtIndex(i))).doubleValue();
    }

    return new BigDecimal(total * 1.5);
}
```

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.

1. Add methods to your custom enterprise object classes.

   Add the code in Listing 3-3 (page 102) to the client-side `Studio.java` file.

   Save the `Studio.java` file.

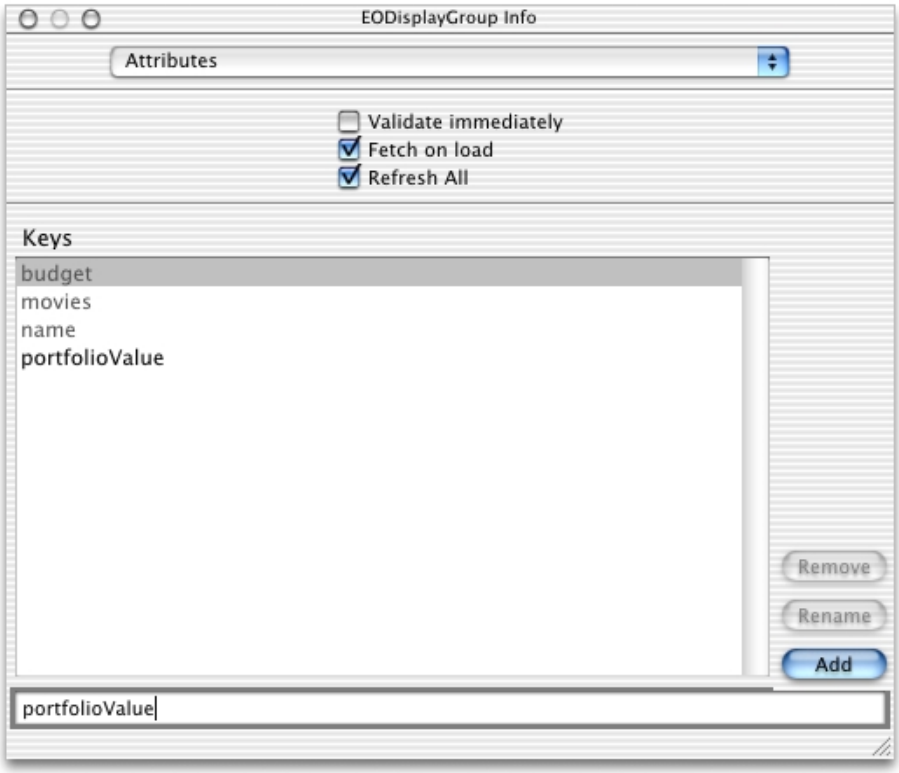
2. Add a custom methods as a display-group properties.

   Open the project's interface (nib) file.

Display the Attributes pane of the Info window for the Studio EODisplayGroup.

Enter the name of the method (`portfolioValue`) you want to use in an association into the text field.

Click Add.



3. Add the necessary user-interface controls.

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

   Using Figure 3-3 as a guide, do the following:

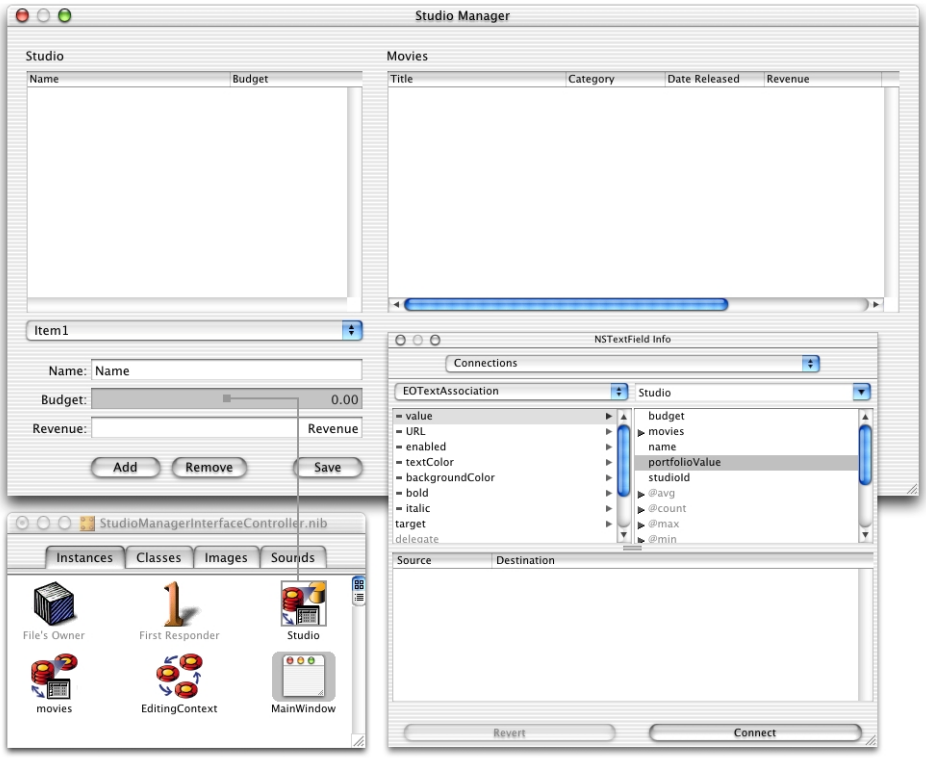
   - Add three text fields to the user interface.
   - Add labels to the left of fields: "Name:", "Budget:", and "Revenue:".
   - Right-justify the Budget and Revenue text fields.

Figure 3-3 Adding elements to the interface

4. Associate interface controls with custom methods.

   Associate the Revenue text field with the `portfolioValue` method:

   - Control-drag from the Revenues text field to the Studio EODisplayGroup.
   - Display the Connections view of the Info window.
   - Choose EOTextAssociation from the pop-up menu.
   - Select `value` in the left column.
   - Select `portfolioValue` in the right column.
   - Click Connect.

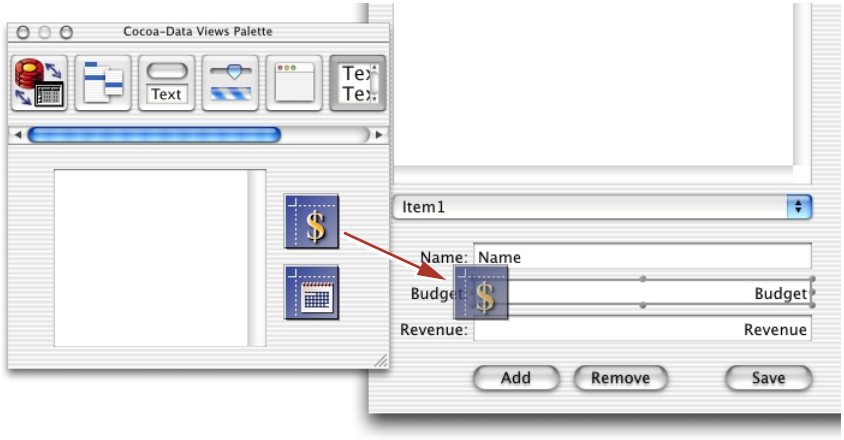


Repeat the process to connect the Name field to Studio's `name` attribute and the Budget field to the `budget` attribute.

5. Add the necessary formatters to the interface controls and choose the format.

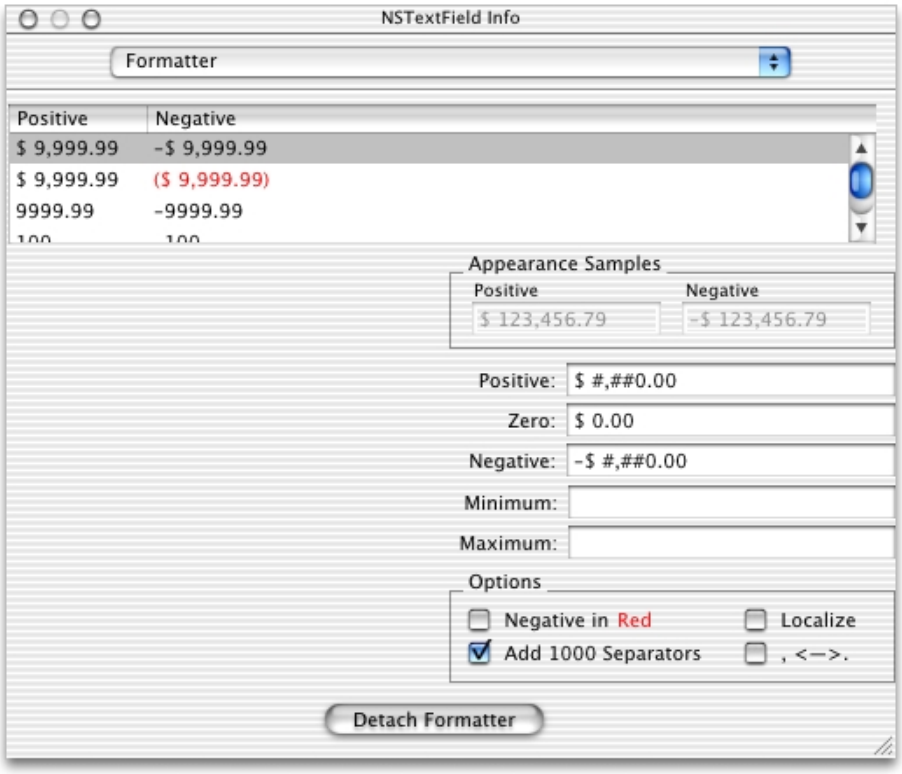
   Currency formatters aren't added automatically, because a field has no way of knowing that it's going to be used to display currency values—it's just connected to a property.

   From the Data Views palette, drag the currency formatter into the Budget text field.



Once you've added the formatter, you can use the Info window to change the display format.

Select the standard currency format.



Repeat the process for the Revenue text field.

6. Save the interface file.
7. Build and test the application on the client.

   (See "Building and Testing Your Application" (page 64) for details.)

Performing Validation

Another behavior 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 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 one shown in Listing 3-4.

Listing 3-4 Studio.java (server and client) - Validation

```
public void validateBudget(Number budget) throws
    NSValidation.ValidationException {
    if (budget.intValue() < 100) {
        throw new NSValidation.ValidationException
            ("A budget cannot be less than $100");
    }
}
```

You use a NSValidation.ValidationException object to tell Enterprise Objects Framework that the current object graph is not cleared to be saved to the database.

Now when a studio buys more movies than it can afford, a panel displaying the message "A budget cannot be less than $100" appears when the user attempts to save the changes to the database.

Validation methods must be of the form `validate`*Attribute*. The `validateBudget` method is invoked by the `validateValueForKey` method, which is part of the EOValidation interface 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

- validateClientUpdate
- validateForSave
- validateForDelete
- validateForInsert
- validateForUpdate

You can find more information on this topic in the book *Enterprise Objects Framework Developer's Guide*.

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 `awakeFromInsertion`. This method is automatically invoked right after your enterprise object class creates a new object and inserts it into an EOEditingContext.

Listing 3-5 shows the implementation of `awakeFromInsertion` in the Studio class. It sets the default value of the budget property to be one million dollars.

Listing 3-5 Studio.java (server and client) - Default values

```
public void awakeFromInsertion(EOEditingContext context) {
    super.awakeFromInsertion(context);
    if (budget() == null) {
        setBudget(new BigDecimal("1000000"));
    }
}
```

When a user clicks the Add Studio button in the StudioManager application, a new record is inserted, with "$1,000,000.00" already displayed as a value in the Budget column.

Invoking Server Methods Remotely

In a Java Client application you may want some methods to execute only on the server. This is particularly the case when security is an issue, but performance can be a reason as well (as when the method consumes a lot of system resources). Objects on the client side of a Java Client application can use two methods to invoke a server method:

- **invokeRemoteMethod:** An enterprise object on the client side can use this method to invoke a method in the corresponding enterprise object on the server. The arguments are the name of the method to invoke and an array of arguments. Before the method is invoked on the server, the current state of the client-side editing context is "pushed" to the server to ensure that the method executes in an identical context. (Note that EODistributedObjectStore has a version of this method that includes a flag as an argument; setting this flag to `false` prevents the client from pushing its editing-context state to the server.)
- **invokeRemoteMethodWithKeyPath:** You can send a message to *any* object on the server with this method, which is defined in EODistributedObjectStore. For more on this method, see the specification for this EODistribution class.

If you want to give studios the ability to buy all of the movies that star a specified actor but consider this a sensitive computation, you can implement a method like the one in Listing 3-6 in the client's `Studio.java`.

Listing 3-6 Studioi.java (client) - Buying all the movies starring a specific talent

```
public void buyAllMoviesStarringTalent(Talent talent) {
    invokeRemoteMethod("clientSideRequestBuyAllMoviesStarringTalent",
    new Class[] {Object.class}, new Object[] {talent});
}
```

The method begins with `clientSideRequest`; this is not accidental. The EODistributionContext object on the server-side EODistribution layer will reject a remote invocation unless it has this prefix *or* its delegate implements the proper delegation methods (see the reference documentation for EODistributionContext or EODistributedObjectStore for more information).

Listing 3-7 shows the invoked method, which is implemented in the server's `Studio.java`.

Listing 3-7 Studio.java (server) - Buying all the movies starring a specific talent

```
public void buyAllMoviesStarringTalent(Talent talent) {
    int i, count;
    NSArray talentMovies;
    EOEnterpriseObject movie, studio;

    talentMovies = talent.moviesStarredIn();
    count = talentMovies.count();
    for (i = 0; i < count; i++) {
        movie =
            (EOEnterpriseObject)(talentMovies.objectAtIndex(i));
        if (!(movies().containsObject(movie))) {
            studio =
                (EOEnterpriseObject)(movie.valueForKey("studio"));
            if (studio != null)
                studio.
                removeObjectFromBothSidesOfRelationshipWithKey
```

```
                (movie,"movies");
            addObjectToBothSidesOfRelationshipWithKey
                (movie,"movies");
        }
    }
}

public void clientSideRequestBuyAllMoviesStarring(Object object) {
    buyAllMoviesStarringTalent((Talent)object);
}
```

Listing 3-8 shows server's `buyAllMoviesStarringTalent` method, which invokes the `moviesStarredIn` method.

Listing 3-8 Talent.java (server) - Buying all the movies starring a specific talent

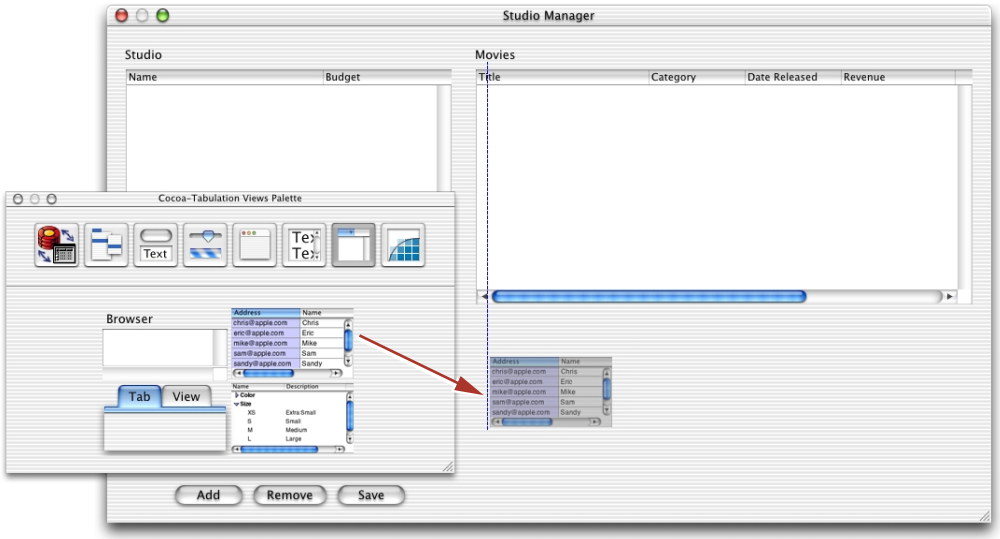
```
public NSArray moviesStarredIn() {
    int i, count;
    NSArray movies;
    NSMutableArray moviesStarredIn;
    EOEnterpriseObject movie;

    moviesStarredIn = new NSMutableArray();
    movies = (NSArray)(movieRoles().valueForKey("movie"));

    count = movies.count();
    for (i = 0; i < count; i++) {
        movie = (EOEnterpriseObject)(movies.objectAtIndex(i));
        if (!(moviesStarredIn.containsObject(movie))) {
            moviesStarredIn.addObject(movie);
        }
    }
    return moviesStarredIn;
}
```

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

1. Add a new table view to your user interface.

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

Drag a table view from the Palette onto your window.



2. Associate the table view columns with enterprise objects.

   Control-drag from the first table view column into the Talent EODisplayGroup.

   Using the value aspect of the EOTableColumnAssociation, connect the table view column to the `firstName` attribute of the Talent EODisplayGroup.

   Using a similar process, connect the second column of the table view to the `lastName` property of the Talent EODisplayGroup.

3. Add a button to the window.

   Drag a button into the window and place it below the Revenue field.

   Give it the title “Buy Movies Starring Selected Talent”.

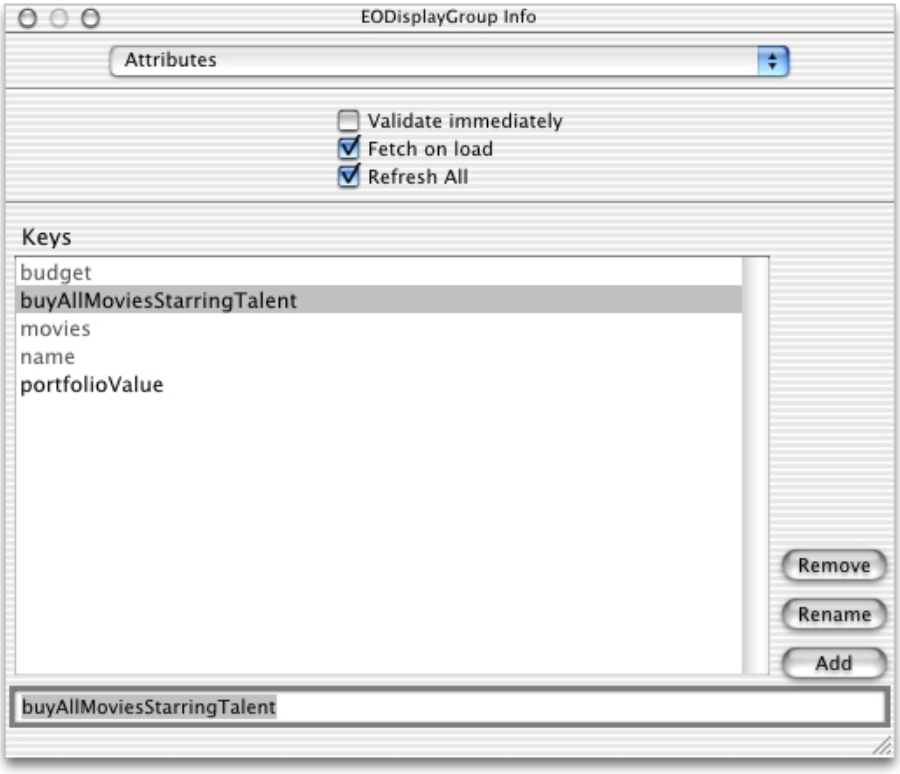
4. Add a method to an EODisplayGroup.

   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 `buyAllMoviesStarringTalent` method.

Display the Attributes pane of the Info window for the Studio EODisplayGroup.

In the text field type the name of the method (`buyAllMoviesStarringTalent`) you want to use in an association.

Click Add.



You can now use the `buyAllMoviesStarringTalent` method in associations.

5. Associate a user interface element with a method.

   Control-drag from the Buy Movies Starring Selected Talent button to the Studio EODisplayGroup.

   In the Connections pane of the Info window, choose EOActionAssociation from the pop-up menu at the top of the left column.

   Select `action` in the left column, and the method you want to connect to (`buyAllMoviesStarringTalent`) in the right column.

Click Connect.

6. Associate a user interface element with the arguments it provides to its method.

   Because the `buyAllMoviesStarringTalent` method takes a Talent object as an argument, you also need to make a connection from the Buy button to the Talent EODisplayGroup.

   Control-drag from the Buy Movies Starring Selected Talent button to the Talent EODisplayGroup.

   In the Info window, 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 `buyAllMoviesStarringTalent` 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.

7. Save the interface.
8. Build and test your application.

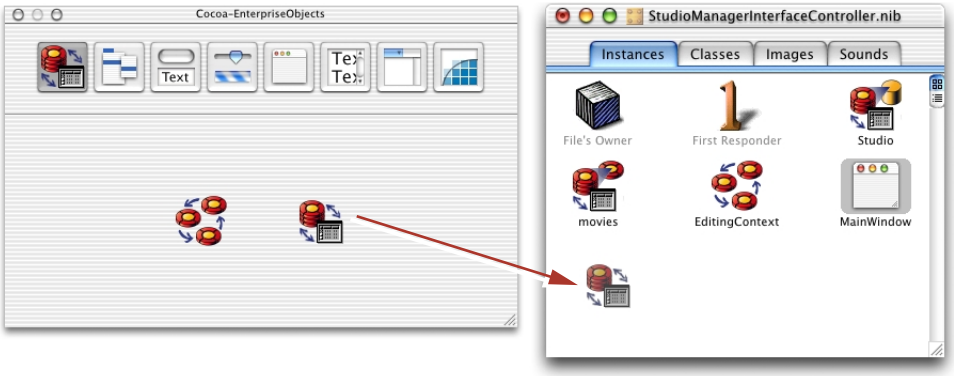
Controlling the User Interface

In Java Client applications you can give the interface controller (implemented in this project in `StudioManagerInterfaceController.java` on the client) a *controller display group*. By creating associations between the controller display group and aspects of user-interface elements, you can use the interface controller to manage various facets of the user interface. In the following steps, you add a method as a property

of the controller display group and bind this method to the `enabled` aspect of the Revenue field through an EOControlAssociation; since this method simply returns `false`, the field is disabled.

1. Add a display group to the nib file.

   Drag a display group from the EnterpriseObjects Palette to the nib file window.



   Double-click the title of the display group to select it.

   Give the display group the name “Controller”.

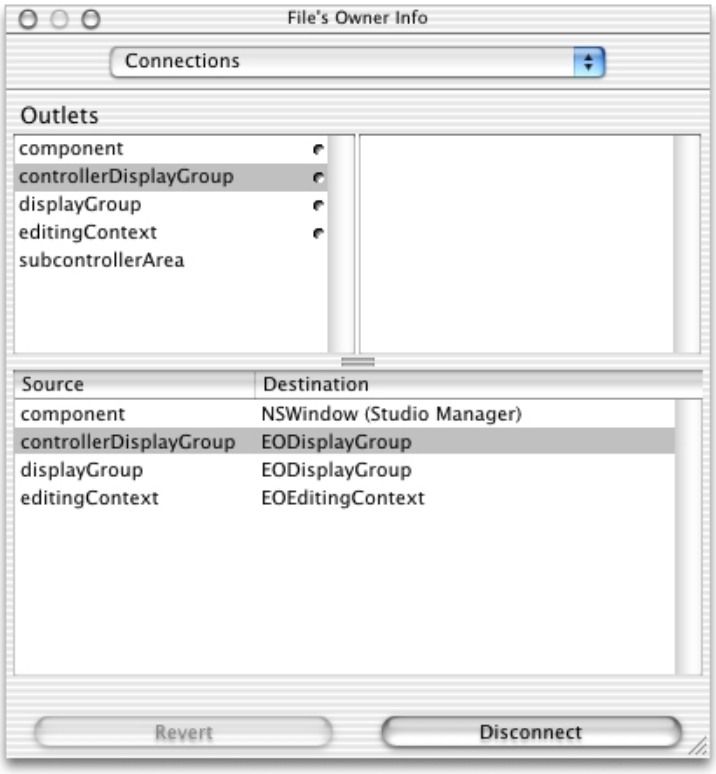
2. Connect the interface controller to its display group.

   As mentioned earlier, the owner of the nib file (File’s Owner) is an instance of the custom EOInterfaceController automatically created by Project Builder. EOIntefaceController has a `controllerDisplayGroup` outlet; you’ll connect the interface controller to this outlet.

   Control-drag from File’s Owner to the Controller icon.

   In the Connections pane of the Info window, select controllerDisplayGroup.

   Click Connect.



3. Add a property to the controller display group.

   Now you'll add the `neverEnabled` method as a property of the controller display group.

   Select the Controller display group in the nib file.

   In the Attributes pane of the Info window, enter `neverEnabled` in the field.

   Click Add.

4. Connect an interface element to a property of the controller's display group.

   Now you'll hook up the field to the display group using an EOActionAssociation to bind its `enabled` aspect to the `neverEnabled` method.

   Control-drag from the Revenue field to the Controller display group.

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

   Select `enabled` in the left column.

   Select `neverEnabled` in the right column.

   Click Connect.

5. Implement the `neverEnabled` method.

   Now that the interface controller, the controller display group, and the Revenue field are interconnected via their outlets and associations, you can implement the method bound to the `enabled` aspect (in `StudioManagerInterfaceController.Java` on the client) as Listing 3-9 shows.

Listing 3-9 neverEnabled method

```
public boolean neverEnabled() {
        return false;
}
```

6. Build, run, and test the application.

Build the project and test the application. The user can copy the contents of the Revenue field but it cannot be written into.

Advanced Tasks

Debugging Java Client WebObjects Applications

It can be difficult to debug Java Client WebObjects applications because these applications have a client side and a server side. Each side runs in a totally different process and in a different virtual machine (VM), so you can't debug the one side by running a debugger for the other side.

Debugging Server Code

To debug the server side of a Java Client application use the standard debugging features of Project Builder. Start the debugger by clicking the Debug button (the spray-can icon). You can use this procedure to perform debugging tasks in all your server side classes.

See the documentation for Project Builder for details on its debugging features.

Debugging Client Code

Project Builder currently provides no support for debugging the client side of a Java Client application. Instead, use the Java debugger `jdb` (included with the JDK) in a shell window.

Once your code has compiled, start up the client application with `AppletViewer` or with the `java` interpreter (see "Running a Java Client Application" (page 66) in the tutorial) with the -debug flag. These tools then print a "password" that you can use later to attach `jdb` to your client application. To attach `jdb`, open another shell and enter the following command:

`jdb -password` *password*

Please refer to the `jdb` documentation for information on setting breakpoints and performing other debugging tasks. As with running an application, your CLASSPATH environment variable has to specify the location of all Java classes used in your application.

If you don't want to attach to a running client application, you can start up `jdb` using `AppletViewer` or the interpreter through a class name, for example:

`jdb sun.applet.AppletViewer` *URL*

`jdb com.webobjects.eoapplication.EOApplication` *-applicattionURL URL*

The advantage of starting up `jdb` like this is that you can set breakpoints before your application is executed; `jdb` stops before it executes the main function of the given class.

Note: Use the -WOAutoOpenInBrowser NO flag when starting up your server application to prevent the client application from automatically launching in your default browser.

Customizing Your Project With Assistants

Project Builder includes several features, including assistants, that you can—and should—use to add Web components, interface-controller subclasses, and client-side interface files to Java Client applications. This is especially true with interface (nib) files; never create a client-side nib file using Interface Builder (as, for instance, by choosing the New Database Interface command from the Document menu).

Adding Interface Controller Subclasses and Nib Files

To add an EOInterfaceController subclass with a new interface file to your client-side subproject

1. Select the Interfaces group in your project file.
2. Chose File > New File.
3. Select WebObjects > Java Client Interface and click Next.
4. Enter the filename for the new EOInterfaceController subclass and select its location.
5. Select the Client target and click Next.
6. Enter class and package names for the new interface class and click Next.
7. Select the options that you want for the new interface file.

   Choose the template and available options that you want your interface to use.
8. Follow the subsequent instructions until completion.

After finishing the assistant, Project Builder will add two files to your project: a source (`.java`) file for the EOInterfaceController subclass and the nib file that is owned by the interface controller.

Note: When you create a Java Client project, the EOInterfaceController subclass and its interface file by default have the same name as your application. If you rename these files, you must make adjustments elsewhere in your project, as described in "Manual Adjustments to Java Client Projects" (page 124).

Adding Web Components (with Interface Controllers)

You can use Project Builder to add a web component containing a WOJavaClientApplet component with a binding to an EOInterfaceController subclass in your project. To create such a Web Component

1. Select the Web Components group in your root (main) project.
2. Choose File > New File.
3. Select WebObjects/Component.

4. Click Next.
5. Enter a the name and location of the new component.
6. Select the Server target.
7. Click Finish.

When you complete these steps Project Builder adds the following to your project:

- a subgroup named after your component in the Web Components group
- a Web Component (`.wo`) containing a `.html` and a `.wod` file
- an `.api` file for the component
- a `.java` file

Manual Adjustments to Java Client Projects

You should always use the Java Client wizards if you can because the files that they generate have characteristics that are important for Java Client applications. These files have various dependencies and assumptions, which you must know about if you decide to create them manually.

- The file's owner class of an Java Client interface (nib) file must be the EOInterfaceController subclass that uses it. It is also very important that the package name of the file's owner class is identical to the package name of the interface controller. So if you change the package of the interface controller, you have to open the interface file in Interface Builder and change the name of the EOInterfaceController subclass used for the file's owner.
- The `interfaceControllerClassName` binding of WOJavaClientApplet used in web components has to be the complete class name of an interface controller, including the full package prefix. If you change the package of the interface controller, you have to change the value of the `interfaceControllerClassName` binding.
- If you change the size of a window in a nib file which is later placed in a WOJavaClientApplet (because the WOJavaClientApplet uses the corresponding EOInterfaceController subclass), you have to modify the size bindings of the WOJavaClientApplet so that the window contents still fit into it.

- You might want to add additional bindings to a WOJavaClientApplet. This component takes standard `java.applet` bindings plus some special Java Client ones. See "The Ingredients of a Java Client Project" (page 45) for more information or refer to the WOJavaClientApplet directory in the WebObjects Java Client examples for a complete list of bindings.

CHAPTER 4

Advanced Tasks

APPENDIX A

Enterprise Objects Framework Concepts

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.

What Is a 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. (WebObjects provides a JDBC adaptor that allows you to connect to any JDBC-compliant database.)

What Are EODisplayGroups and EOEditingContexts?

EODisplayGroup

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. The EODisplayGroup that's created when you drag an entity from EOModeler into Interface Builder is actually a compound object that consists of both an EODisplayGroup and an EODatabaseDataSource.

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*.

What Is an Association?

In the tutorial, when you made a connection from the pop-up list to an EODisplayGroup, you formed an association. Associations were also involved when you created a table view by dragging an entity from EOModeler into Interface Builder—the associations were formed for you as a by-product of dragging in the entity.

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 `titles` aspect for the EOValueSelectionAssociation to display all of the class keys whose values you could choose to display in the pop-up list. EOValueSelectionAssociation also has several other aspects: `selectedTitle`, `selectedIndex`, `selectedObject`, and `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*.

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.

Adding Behavior to Enterprise Objects

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

- 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" in the *Enterprise Objects Framework Developer's Guide*.

Glossary

adaptor, database A mechanism that connects your application to a particular database server. For each type of server you use, you need a separate adaptor. WebObjects provides an adaptor for databases conforming to JDBC.

adaptor, WebObjects A process (or a part of one) that connects WebObjects applications to an HTTP server.

Application Kit The Application Kit is a framework containing all the objects you need to implement your graphical, event-driven user interface: windows, panels, buttons, menus, scrollers, and text fields. The Application Kit handles all the details for you as it efficiently draws on the screen, communicates with hardware devices and screen buffers, clears areas of the screen before drawing, and clips views.

application object An object (of the WOApplication class) that represents a single instance of a WebObjects application. The application object's main role is to coordinate the handling of HTTP requests, but it can also maintain application-wide state information.

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 also **entity**; **relationship**.

business logic The rules associated with the data in a database that typically encode business policies. An example is automatically adding late fees for overdue items.

CGI A standard for interfacing external applications with information servers, such as HTTP or Web servers. Short for Common Gateway Interface.

class In object-oriented languages such as Java, a prototype for a particular kind of object. A class definition declares instance variables and defines methods for all members of the class. Objects that have the same types of instance variables and have access to the same methods belong to the same class.

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 also **attribute**.

component An object (of the WOComponent class) that represents a web page or a reusable portion of one.

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.

Direct to Java Client A WebObjects development approach that can generate a Java Client application from a model.

Direct to Java Client Assistant A tool used to customize a Direct to Java Client application.

Direct to Web A WebObjects development approach that can generate a HTML-based Web applications from a model.

Direct to Web Assistant A tool that used to customize a Direct to Web application.

Direct to Web template A component used in Direct to Web applications that can generate a web page for a particular task (for example, a list page) for any entity.

dynamic element A dynamic version of an HTML element. WebObjects includes a list of dynamic elements with which you can build your component.

enterprise object A Java object that conforms to the key-value coding protocol and whose properties (instance data) can map to stored data. An enterprise object brings together stored data with methods for operating on that data. See also **key-value coding**; **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 also **attribute**; **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.

EOModeler A tool used to create and edit models.

faulting A mechanism used by WebObjects to increase performance whereby destination objects of relationships are not fetched until they are explicitly accessed.

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

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 also **primary key**; **relationship**.

HTML-based application approach A WebObjects development approach that allows you to create HTML-based Web applications.

inheritance In object-oriented programming, the ability of a superclass to pass its characteristics (methods and instance variables) on to its subclasses.

instance In object-oriented languages such as Java, an object that belongs to (is a member of) a particular class. Instances are created at runtime according to the specification in the class definition.

Interface Builder A tool used to create and edit graphical user interfaces like those used in Java Client applications.

Java Browser A tool used to peruse Java APIs and class hierarchies.

Java Client A WebObjects development approach that allows you to create graphical user interface applications that run on the user's computer and communicate with a WebObjects server.

Java Foundation Classes A set of graphical user interface components and services written in Java. The component set is known as Swing.

JDBC Stands for "Java Database Connectivity." An interface between Java platforms and databases.

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

key An arbitrary value (usually a string) used to locate a datum in a data structure such as a dictionary.

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 application.

key-value pair See **key-value** coding.

locking A mechanism to ensure that data isn't modified by more than one user at a time and that data isn't read as it is being modified.

look In Direct to Web applications, one of three user interface styles. The looks differ in both layout and appearance.

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 also **relationship**.

method In object-oriented programming, a procedure that can be executed by an object.

model An object (of the EOModel class) 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.

Model-View-Controller An object-oriented programming paradigm in which the functions of an application are separated into the special knowledge (Model objects), user interface elements (View objects), and the interface that connects them (the Controller object).

Monitor A tool used to configure and maintain deployed WebObjects applications capable of handling multiple applications, instances, and application servers at the same time.

object A programming unit that groups together a data structure (instance variables) and the operations (methods) that can use or affect that data. Objects are the principal building blocks of object-oriented programs.

primary key An attribute in an entity that uniquely identifies rows of that entity. For example, the Employee entity can contain an EmpID attribute that uniquely identifies each employee.

Project Builder A tool used to manage the development of a WebObjects application or framework.

property In Entity-Relationship modeling, an attribute or relationship. See also **attribute**; **relationship**.

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

referential integrity The rules governing the consistency of relationships.

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 although 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 also **to-one**; **to-many**; **many-to-many**; **primary key**; **foreign key**.

reusable component A component that can be nested within other components and acts like a dynamic element. Reusable components allow you to extend the WebObject's selection of dynamically generated HTML elements.

request A message conforming to the Hypertext Transfer Protocol (HTTP) sent from the user's Web browser to a Web server that asks for a resource like a Web page. See also **response**.

request-response loop The main loop of a WebObjects application that receives a request, responds to it, and awaits the next request.

response A message conforming to the Hypertext Transfer Protocol (HTTP) sent from the Web server to the user's Web

browser that contains the resource specified by the corresponding request. The response is typically a web page. See also **request**.

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

rule In the Direct to Web and Direct to Java Client approaches, a specification used to customize the user interfaces of applications developed with these approaches.

Rule Editor A tool used to edit the rules in Direct to Web and Direct to Java Client applications.

session A period during which access to a WebObjects application and its resources is granted to a particular client (typically a browser). Also an object (of the WOSession class) representing a session.

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.

target A blueprint for building a product from specified files in your project. It consists of a list of the necessary files and specifications on how to build them. Some common types of targets build frameworks, libraries, applications, and command-line tools.

template In a WebObjects component, a file containing HTML that specifies the overall appearance of a web page generated from the component.

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.

transaction A set of actions that is treated as a single operation.

uniquing A mechanism to ensure that, within a given context, only one object is associated with each row in the database.

validation A mechanism to ensure that user-entered data lies within specified limits.

WebObjects Builder A tool used to graphically edit WebObjects components.

Glossary