WebObjects Java Client Programming Guide

(Legacy)

Tools > WebObjects

2005-08-11

Contents

Figures, Tables, and Listings

Introduction to WebObjects Java Client Programming Guide

Important: The information in this document is obsolete and should not be used for new development.

Note: This document was previously titled *Java Client Desktop Applications*.

WebObjects recognizes the need for distributed, three-tier application solutions with more complex, rich, and responsive user interfaces than HTML allows. So, in addition to HTML-based WebObjects applications, you can also write Java-based WebObjects desktop applications that use Swing for the user interface. The client part of these applications runs as real desktop applications in the client's Java virtual machine. This feature of WebObjects is called Java Client.

WebObjects Java Client is a three-tier network application solution that allows you to develop platform-agnostic desktop applications with database access and rich user interfaces. Java Client applications are WebObjects applications: They share much of their API with traditional HTML-based WebObjects applications such as the Enterprise Object technology for database access, the WebObjects framework for session management, and the rule system, which enables rapid development and provides a flexible rule-based approach to application development.

Note: This book describes WebObjects 5.2. Future versions of WebObjects may include API changes and other changes that affect the tutorials, sample code, and concepts described herein.

This book introduces you to Java Client by first presenting key concepts such as architecture, enterprise objects, client-server communication, object distribution, the Model-View-Controller paradigm, and rule-based application development. Then, you are led through the development of simple yet practical tutorials that introduce you to the WebObjects developer tools and the features of Java Client. Finally, the book provides a number of task-specific chapters that teach you how to add to applications features like access controls, custom menu items, and sophisticated user interfaces.

There are two starting points in Java Client development—the Direct to Java Client project type and the Java Client project type. This book teaches you how to build Java Client applications starting with the Direct to Java Client approach. This approach reduces the amount of code you need to write and lets you take advantage of some of the best features of WebObjects such as the rule system and rule-based rapid development. And you can easily integrate all aspects of the nondirect approach (such as hand-built user interfaces) into the direct approach for maximum flexibility.

Some of the customizations you'll perform in the tutorials—such as building user interfaces in Interface Builder—teach you just about everything you need to know to build strictly nondirect Java Client applications. So if you're an experienced Java Client developer, don't think that this book isn't for you. By learning how to leverage the features of the Direct to Java Client approach, you'll learn how to build better Java Client applications.

Who Should Read This Book

This book is intended for a wide variety of audiences, including

- new WebObjects developers
- new Web developers
- WebObjects HTML developers
- experienced WebObjects Java Client developers

This book assumes that you have some background in object-oriented programming, specifically in Java. Because WebObjects is most valuable when used to provide database connectivity to distributed applications, a basic understanding of relational databases is assumed throughout the book.

The book, however, does not assume any prior knowledge of WebObjects. Although you'll better understand the advanced concepts in Java Client if you've developed HTML-based WebObjects applications, this knowledge isn't necessary to be a successful Java Client developer.

If you're new to WebObjects development, you may find the book *WebObjects Web Applications Programming Guide* useful when learning Java Client as it provides an introduction to the WebObjects tools and to common WebObjects programming techniques and concepts.

The most important part of any type of WebObjects application is the data model. If data modeling is a new concept for you, read *EOModeler User Guide* to learn how to build data models that describe the object-relational mapping model that forms the core of your application.

Finally, the book *WebObjects Direct to Web Guide* helps you better understand the rule system and the dynamic user-interface generation it provides. Familiarity with these concepts will help you grasp the mechanics of Direct to Java Client.

Organization of This Document

If you're new to Java Client, start with the chapter "Java Client Concepts" (page 35) to familiarize yourself with the Java Client architecture and to learn about the fundamental objects used in a Java Client application, especially enterprise objects. Then, move on to "Building a Simple Application" (page 47) to learn how to set up a simple database and build a Direct to Java Client application that accesses it.

If you've had some experience with Java Client, you may want to start with the chapter "Enhancing the Application" (page 103), which covers topics like business logic partitioning, user interface customization, and custom actions. Or, if you're already comfortable with these topics, you may want to consult the task chapters, beginning with "Restricting Access to an Application" (page 177), to learn how to change application flow, integrate Interface Builder files into Direct to Java Client applications, write custom controller classes, and extend applications in other ways.

This book presents the topic of Java Client applications in a way different from previous books and tutorials in the WebObjects documentation suite. In the past, Direct to Java Client and Java Client were considered as two different approaches to Java Client development. But it is more correct to simply understand them as different starting points in Java Client development.

This book encourages you to begin development with the Direct to Java Client project type and it presents aspects of the nondirect approach as customization techniques for applications developed from the Direct to Java Client starting point. You are strongly encouraged to begin development with the direct approach and to use nondirect interfaces within it to leverage the best of both worlds. See "Java Client Development" (page 23) for more information on this topic.

Java Client in WebObjects

The Java Client feature of WebObjects has many parts. It includes these frameworks:

- JavaEOGeneration
- JavaEOApplication
- JavaEODistribution
- JavaEORuleSystem
- JavaEOInterface

The product also includes the following example projects that are specific to Java Client:

- JCDiscussionBoard
- JCRealEstate
- JCRealEstatePhotos
- JCAuthentication
- JCMovies
- JCPointOfSale
- JCRentalStore
- JCStudios
- JCEntityViewer
- JavaClientLauncher

In addition to this book, documentation for Java Client can be found in the API reference for these packages:

- `com.webobjects.eogeneration`
- `com.webobjects.eogeneration.rules`
- `com.webobjects.eoapplication`
- `com.webobjects.eoapplication.client`
- `com.webobjects.eointerface`
- `com.webobjects.eointerface.swing`
- `com.webobjects.eodistribution`
- `com.webobjects.eodistribution.client`

Java Client Features

If you're looking for a three-tier Java application platform with robust data access, rapid development tools, and powerful, innovative customization capabilities, WebObjects Java Client is the perfect solution. Consider the features it offers.

Better User Experience

Java Client applications differ from HTML-based WebObjects applications in that the user interface is built on Sun's JFC/Swing classes, rather than on HTML. This lets Java Client applications take advantage of the rich user interface elements the Swing toolkit offers. The primary reason for choosing to build a WebObjects application using Java Client is the need for a rich, more interactive user interface.

Rich user interfaces let you build more complex and interactive applications than HTML allows. As the user interface becomes more robust, it is easier to display and manipulate complex data. The more active feel of desktop applications gives users the ability to work more efficiently: Desktop applications feel like they are closer to the data store.

Object Distribution

Java Client is built on the paradigm of object distribution. It distributes enterprise objects between an application server and one or more clients—Java applications or applets. The developer controls how this distribution occurs.

In all multitier network applications, it's vitally important that the developer has control over where the business logic sits. Some information such as credit card numbers and passwords are important elements of business logic and should not be sent to the client. Likewise, certain algorithms represent confidential business logic and should live only on the application server. By partitioning your business logic into client-side and server-side classes, you can improve performance and secure business rules and legacy data.

In pure Java applications, object distribution is crucial in protecting business rules. Since Java bytecode is easily decompiled, it's important that you have control over the objects that live on the client. Object distribution, coupled with remote method invocation, lets you build secure, high-performance applications.

The Best of WebObjects

As with any type of WebObjects application, Java Client gives you a lot for free. Its tight integration with the Enterprise Object technology takes care of many basic database access tasks for you. Without writing a single line of code, Java Client allows you to connect user interface widgets to database actions such as saving, retrieving, reverting, undoing, adding objects, and editing objects. Furthermore, Java Client's integration with Enterprise Objects abstracts development above the need to ever write a line of SQL. And the development tools you use to build Java Client applications let you build complex user interfaces in Swing without writing any code.

It is the WebObjects philosophy that the technology should take care of all the tasks fundamental to three-tier applications: database access, user interface coding, deployment, and client-server communication. That way, you can focus on writing business logic that best leverages the powerful data access mechanisms all WebObjects applications offer.

Deployment Options

The client-side application of WebObjects Java Client applications runs on any Java Runtime Environment (JRE) 1.3.1 or later system. The server-side application runs on any supported WebObjects deployment server, which includes many J2EE servers. Since the Java Client architecture isolates the application logic from any particular data access mechanism, you have the flexibility to use many types of JDBC and JNDI data sources regardless of the deployment platform.

Rapid Application Development

In addition to supplying powerful data modeling, project development, and interface building tools, Java Client includes a sophisticated rapid-development environment based on the WebObjects rule system.

The Java Client rapid-development starting point, called Direct to Java Client, generates application user interfaces by analyzing your application’s data model. Direct to Java Client allows you to immediately see how changes in your data model affect your application’s user interface.

Direct to Java Client lets you focus on writing custom business logic and provides customization techniques that allow you to build sophisticated user interfaces without writing any code. Best of all, Direct to Java Client applications are completely integrated with the Enterprise Object technology, so they take full advantage of the rich data access and persistence mechanisms that technology offers. And applications built from the Direct to Java Client starting point benefit from all aspects of applications built from the nondirect Java Client starting point.

When to Choose Java Client

Java Client is a great technology for developing and deploying desktop applications with powerful database access in controlled network environments where the end users are known and are willing to install parts of the client application. It is not ideal, however, for use in uncontrolled Internet environments or for high-traffic websites. Typically, Java Client applications are practical only in *intranet* environments.

Consider the case of a software company’s bug-tracking system. Perhaps the company wants to give premium support customers access to the system through a Java Client application. These customers are assumed to be knowledgeable users and would have no problem downloading and installing certain parts of the client application. However, providing the client application as a desktop application from the company’s main website to a large number of novice end users would be impractical due to the support those users would need installing and maintaining a current version of the client application.

When deployed as desktop applications, Java Client applications have special deployment requirements because part of the application runs on the user’s computer. Unlike HTML-based applications, it is not enough to have a browser application to run a Java Client application as a desktop application. You either need to install the client-side application on user computers, which requires system administration, or users need to download the client-side application every time they want to use it. This makes Java Client applications too complex for the average Internet application user who expects to type a URL in a browser and enter an application within seconds of hitting the website.

However, you can also deploy Java Client applications as applets that run in browsers. This alleviates many of the issues encountered when running Java Client applications as desktop applications as users don't need to download or install the client application. However, applets introduce other usability and deployment issues, which are discussed in "Deployment Options" (page 169). And, starting with WebObjects 5.2, support for deploying Java Client applications as applets is deprecated in favor of using Java Web Start.

Starting with WebObjects 5.1, the Java Client Class Loader eases deployment and improves usability, thereby alleviating many of the issues regarding application distribution and maintenance. See "More About the Java Client Class Loader" (page 63) for more information.

Starting with WebObjects 5.2, all Java Client applications take advantage of Java Web Start. This technology eases client-side application deployment and usability by providing caching and other mechanisms to the client. It requires no installation or upgrades on the part of the user. It requires only the presence of the JRE and the Java Web Start Application Manager on the client. In short, Web Start expands the possible user base of Java Client applications by providing transparent application installation and upgrades.

That said, in deciding to use Java Client, you should evaluate the technology with these criteria in mind: portability, performance, network environment, administration, security, and user experience.

- **Portability.** Java Client applications are 100% Pure Java applications, requiring a JRE 1.3 or later system. Java Client applications running in Mac OS X take advantage of platform-specific interface features such as the global menu bar and the dirty window marker without compromising platform independence.
- **Performance.** After the initial download of Java classes to the client, Java Client applications don't exchange large chunks of data between client and server. Rather, compact business objects are exchanged over the network. Also, the Java Client architecture separates the user interface layer from the data exchange layer, so data flows across the network independent of user interface data.

  For example, in an HTML-based application, switching panes in a tab view requires a round trip to the server to fetch more data or user interface information. However, in Java Client, the client application usually has no need to contact the server for these types of user interface actions. This allows Java Client applications to scale well, and a WebObjects application server should scale just as well serving Java Client applications or HTML-based applications.
- **Network environment.** Java Client applications can be deployed across the Internet; they are not inherently constrained to intranet environments. However, they are not appropriate for high-volume, high-visibility websites because of the long initial download and other system administration requirements (including the presence of JRE 1.3 or later).
- **System administration.** The presence of JRE 1.3 or later is not ubiquitous amongst desktop operating systems. Mac OS X includes JRE 1.3 out of the box; JRE 1.3 is not available for Mac OS 9 or earlier versions; Sun provides the JRE for Windows platforms, but it does not ship in the box; JRE 1.3 is available for many UNIX platforms. So, while the JRE is widely available, it must often be downloaded and installed by the end user. You should evaluate your target market, keeping in mind that installing the JRE will frustrate some customers.
- **Security.** If you take careful steps to partition your business logic and implement the appropriate security mechanisms (delegates), Java Client applications offer security equal to that of HTML-based applications. By default, Java Client uses HTTP as the transport protocol between client and server, but it can be replaced with another, more secure protocol such as SSL.
- **Client-side processing.** Web applications do the majority of their processing on the server, while Java Client moves much of an application's processing to the client. This reduces the amount of client-server communication considerably, making Java Client applications much quicker than their Web counterparts for many operations.

- **User experience.** All the preceding criteria affect user experience in some way. If your application demands a rich user interface, the manipulation of complex data, and long sessions, Java Client is an excellent choice.

Java Client Development

There are two starting points in Java Client development represented by two Project Builder project types: Direct to Java Client and Java Client. *You should always start with the Direct to Java Client project type.* The nondirect project type gives you almost no advantages—you write more code, the application is less dynamic, and maintenance costs are much higher. And you can use all the features of nondirect Java Client in Direct to Java Client applications, so you don't lose anything by starting with the Direct to Java Client project type. So unless you know that your application will not gain anything from using the rule system and dynamic user-interface generation, always choose the direct approach when building a Java Client application.

Without customizations, the fundamental difference between the two project types is that Direct to Java Client makes use of the rule system and nondirect Java Client does not.

You can think of the relationship this way: An uncustomized nondirect Java Client application is a completely customized Direct to Java Client application that doesn't use the rule system for building user interfaces or managing the basic tasks of the client application such as application startup. Whereas the user interface in uncustomized Direct to Java Client applications is generated dynamically at runtime and can include static, hand-built user interfaces, the user interface in nondirect Java Client applications is always static and built by hand.

Perhaps the most significant difference between the two starting points is that Direct to Java Client provides a rapid development environment that is useful both for prototyping applications and for building full-featured, usable applications. When you start with the nondirect approach, you get almost nothing for free—you have to build all the user interfaces for the application by hand. This book highly recommends that you begin with the Direct to Java Client project type and use elements of the nondirect project type within it if necessary.

If you need the precise user-interface customization that the nondirect approach allows, it's much easier to integrate a custom interface file in a Direct to Java Client application than to develop a completely custom Java Client application (though this is possible and supported). That way, you get the best of both worlds: the advantages of Direct to Java Client and the advantages of custom interfaces built with the nondirect approach.

The primary advantage of Direct to Java Client is that it's not necessary to write source code to generate or manage all of an application's user interface. This allows you to focus on writing business logic instead. The direct approach lets you manage user interfaces without writing much source code and offers a number of alternative mechanisms to customize user interfaces:

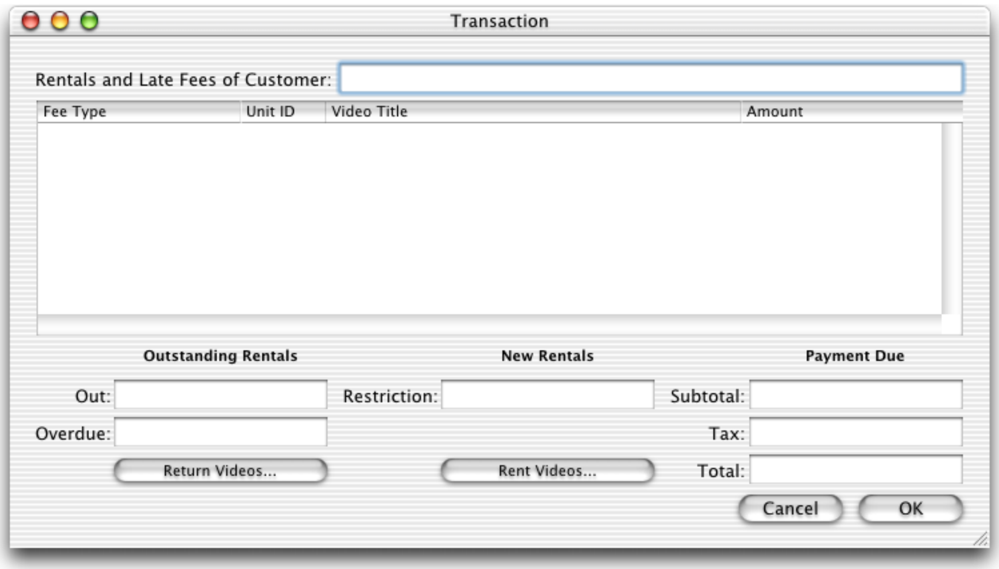
- Direct to Java Client Assistant (tool)
- custom rules (rule system)
- freezing XML (custom interface)
- freezing nib files (custom interface)
- using custom controller classes (custom code)
- using the controller factory programmatically

This book covers all of these customization methods.

The user interfaces for the two staring points to Java Client development each have a particular character. However, keep in mind that it's possible to customize each type of interface to look like the other.

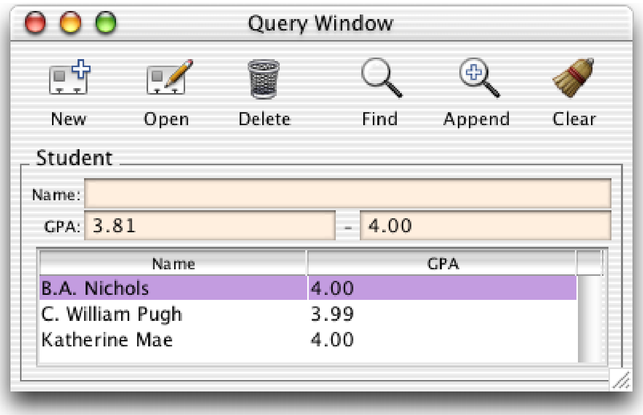
Typically, user interfaces built in Interface Builder for nondirect Java Client applications or for use as frozen interface files in Direct to Java Client applications resemble Figure I-1 (page 24).

Figure I-1 A custom Java Client interface



The dynamic user-interface generation provided in Direct to Java Client applications yields interfaces that resemble Figure I-2. However, advanced Direct to Java Client applications are likely to include other, nondynamically generated user interfaces such as custom controller classes or frozen interface files built in Interface Builder.

Figure I-2 A typical Direct to Java Client application

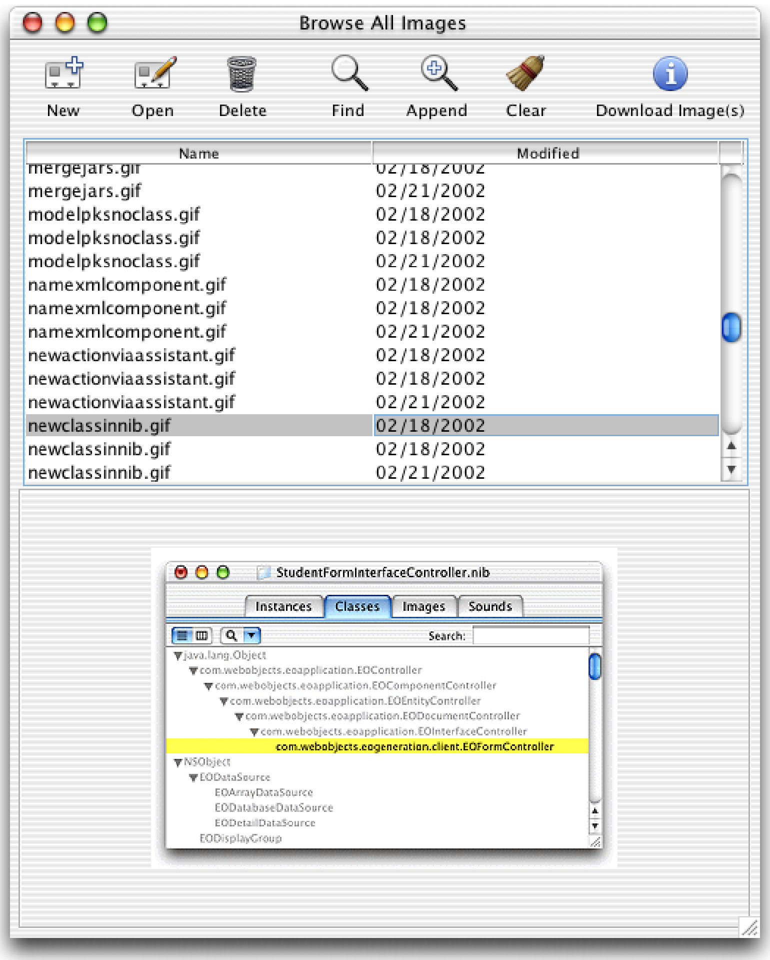


GPA: 3.81 - 4.00

Name	GPA
B.A. Nichols	4.00
C. William Pugh	3.99
Katherine Mae	4.00

Figure I-3 shows dynamically generated user interfaces that make use of custom controller classes, custom rules, and programmatic invocations of the controller factory.

Figure I-3 Dynamically generated user interface



Name	Modified
mergejars.gif	02/21/2002
modelpksnoclass.gif	02/18/2002
modelpksnoclass.gif	02/18/2002
modelpksnoclass.gif	02/21/2002
namexmlcomponent.gif	02/18/2002
namexmlcomponent.gif	02/18/2002
namexmlcomponent.gif	02/21/2002
newactionviaassistant.gif	02/18/2002
newactionviaassistant.gif	02/18/2002
newactionviaassistant.gif	02/21/2002
newclassinnib.gif	02/18/2002
newclassinnib.gif	02/18/2002
newclassinnib.gif	02/21/2002

```
▼java.lang.Object
  ▼com.webobjects.eoapplication.EOController
    ▼com.webobjects.eoapplication.EOComponentController
      ▼com.webobjects.eoapplication.EOEntityController
        ▼com.webobjects.eoapplication.EODocumentController
          ▼com.webobjects.eoapplication.EOInterfaceController
              com.webobjects.eogeneration.client.EOFormController
▼NSObject
  ▼EODataSource
      EOArrayDataSource
      EODatabaseDataSource
      EODetailDataSource
    EODisplayGroup
```

Direct to Java Client simplifies many parts of the development process and facilitates the addition of features such as localization, data access, and data model synchronization. The direct approach to Java Client is a great way to start developing Java Client applications because it allows you to rely on the rule system to dynamically generate user interfaces. Dynamically generated user interfaces are more flexible with regard to changes made in your data model than are static interfaces and provide other advantages as shown in Table I-1 (page 25).

Table I-1 Comparison of static and dynamic user interfaces

	Static interfaces	Dynamic interfaces
Tools and techniques used to build	Interface Builder and raw Swing.	Assistant, XML freezing, Interface Builder files, custom controller classes, controller factory invocations.

	Static interfaces	Dynamic interfaces
Development speed	Moderate to slow depending on user interface design.	Rapid. User interfaces are automatically generated but are also easily customizable.
User interface synchronization with data model	Difficult. User interface not synchronized with data model once user interface building begins.	Synchronization happens throughout much of the customization process.
Localization	Must use different interface files.	Mostly automatic using the rule system.
Maintenance	More frozen code and frozen interface elements to manually maintain.	Applications are easier to maintain and bring forward.

If you decide to start development with the nondirect Java Client approach, you should keep in mind that your application will be harder to bring forward and maintain than an application started with the Direct to Java Client approach. The maintenance costs are higher for a number of reasons:

- you write more code
- you have more frozen interface pieces
- you need multiple versions of the same interface file for each language and platform
- it's harder to synchronize the user interface with changes in data models

So while you can write nondirect Java Client applications, the Direct to Java Client approach helps you build applications that are far easier to bring forward and maintain. You'll also find that application development time is significantly reduced with the direct approach.

Database Access

WebObjects applications gain much of their usefulness by interacting with data stores, and the Enterprise Object technology is the mechanism by which WebObjects applications interact with data stores.

The Enterprise Object technology is responsible for

- communicating with the data source
- representing data fetched from the data source in enterprise objects
- managing the graph of enterprise objects
- mediating between the object graph and user interfaces
- providing application utilities to Java Client applications
- managing object distribution across networks to Java clients

Enterprise Objects is introduced in more detail in "Enterprise Objects" (page 35).

Also see "Related Documents" (page 27) to learn how to access the WebObjects API reference and other documents on the Enterprise Object technology.

Java Client and Other Multitier Systems

There are many distributed multitier Java-based architectures on the market today. So how do they compare to WebObjects Java Client?

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 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. In addition, this architecture doesn't scale; it is expensive to move data over the channel to the client. Also, client JDBC applications access the data source directly—there is no server layer to validate data or control access to the data source.

JDBC three-tier applications (with CORBA as the transport) are a big improvement over client JDBC applications. In this architecture, the client can be thin since all that is required on the client side are the Java Foundation Classes (JFC), nonsensitive custom code (usually for managing the user interface), and CORBA stubs for communicating with the server. Sensitive business logic and database connection logic 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 real objects on the server, each client request for an attribute is forwarded to the server, causing a separate round trip. 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.

The Java Client architecture, however, scales well since real, fully functional data objects are copied to the client and round trips are made to the server only for database commits and new data fetches. Also, Java Client applications are designed to leverage custom business logic that lets you control which business objects are sent to the client and lets you validate data from the client before it's committed to the data store (the server ultimately determines what data is committed).

See Also

The following documents provide more information on developing applications with WebObjects:

- *WebObjects Overview* provides an introduction to the technologies in WebObjects.
- *WebObjects Web Applications Programming Guide* provides an introduction to WebObjects HTML development.
- *EOModeler User Guide* describes EOModeler, the tool you use to create data models in WebObjects.
- *Enterprise JavaBeans* describes how to build Enterprise JavaBeans–based applications and how to use third-party beans in WebObjects applications.
- *WebObjects J2EE Programming Guide* describes how you can leverage WebObjects components in JSP-based applications. It also describes how to deploy WebObjects applications as servlets.

- *WebObjects Web Services Programming Guide* describes how to build Web Service providers using WebObjects and how to adapt WebObjects applications to consume Web services.

You can find further documentation for WebObjects and Java Client in three places:

- Project Builder's Developer Help Center, accessible through the Help menu
- Apple's WebObjects documentation site: http://developer.apple.com/documentation/WebObjects
- the WebObjects CD-ROM, which contains the WebObjects API reference, various documents in HTML and PDF, examples, what's new, and legacy documentation

Development Process Overview

Direct to Java Client is composed of many complex technologies: the rule system, Enterprise Objects, the WebObjects frameworks, and Java Swing, to name a few. Once you have a basic understanding of these technologies, however, you'll be able to build full-featured applications at a rapid rate. This chapter provides a recommended workflow and provides a narrative of the Java Client development process.

You might want to refer back to this chapter as you progress through the rest of the book.

Workflow

A good workflow is essential to successful WebObjects application development. The recommended workflow in any WebObjects application is as follows:

1. Write specifications for the application.
2. Design and build the enterprise object models in EOModeler.
3. Test the models with Direct to Web or Direct to Java Client.
4. Refine the models appropriately.
5. Define access levels.
6. Design application flow.
7. Identify tasks.
8. Implement.

These steps are expanded on in the following sections.

Write Specifications

A specification document for an application includes both high-level summaries of the application's requirements and goals, as well as low-level implementation details. A specification should minimally answer these questions:

- What is the purpose of the application (high-level summary)?
- Whom does the application help (audience)?
- What are the chief goals of the application (in order of priority)?
- What problems will the application address?

- What won't the application do (limitations)?
- What are the target platforms for the application (system requirements)?
- How will the application be installed?
- How will users update the client application?
- What are the performance requirements of the application?
- What are the application's features (list the key items of each feature)?
- How is the application designed?
- What are the key user interface elements?
- What are the application's key algorithms?
- How will the application be tested?

Model the Data

The work necessary in this step differs greatly depending on your data source. If the data for the application already exists, you can use EOModeler's reverse engineering feature to build an EOModel from an existing data source. All that you need to do is set delete rules and optionality for relationships and determine which attributes are class properties or client-side class properties.

However, if you are starting from the beginning, you need to design the database before making model-specific adjustments. Specifically, you need to determine:

- the database tables (model entities)
- the database columns (model attributes)
- the relationships between tables

Then, you can customize the model with Enterprise Objects–specific settings such as optionality, delete rules, and class properties. You should try to follow these data modeling recommendations:

- The model's relationships should be modeled in both directions to support robust searching capabilities (each relationship should have an inverse relationship).
- The model should not include any stored procedures or derived attributes in order to preserve data source-independence.
- Some of the model's relationships should be mandatory to enforce business logic rules.
- Generally, the model should provide a solid foundation for the application's business logic.
- The application should rely on the automatic primary-key generation features of Enterprise Objects.

Test and Refine Models

The WebObjects project types Direct to Web Application and Direct to Java Client Application are good for testing your model. While EOModeler performs consistency checks on your model when you save it, it doesn't guarantee that the model will actually work well. In this step, create either a Direct to Web Application or a Direct to Java Client Application, select your model or models, and experiment with adding data to your data source with each application.

If you encounter abnormal behaviors, refine the models, build and run the application again, and continue to test. Iterate through this process until you have a well-designed model.

Define Access Levels

Defining different access levels in your application is not a necessary part of the design process, but thinking about it early saves you time when it's time to implement access controls. In this task, you should define the groups of users and what if any parts of the application should be restricted to a certain group or groups.

Identify Tasks

Although your application will ultimately be composed of hundreds of tasks, it's important to identify the application's primary tasks. Among other things, this helps you divide development work. In the JCRealEstatePhotos example, the primary tasks include

- importing photos
- searching for photos
- downloading photos

Within each primary task are many subtasks. The importing photos task includes subtasks such as choosing the photos to import and assigning them to a particular listing. The searching for photos task requires writing a user interface so users can provide search criteria, generating qualifiers from the search criteria, performing the fetch, and returning the results. The downloading photos phase includes adding photos to a downloads basket, selecting a downloads location, and performing the download.

In Direct to Java Client development, the most important part when identifying the application's tasks is identifying user interfaces that Direct to Java Client doesn't automatically provide you. This lets you plan development cycles to write nib files or custom controllers for custom user interfaces.

Design Application Flow

The decisions you make about application flow will most directly affect your application's users. You have to think about how users will access the tasks you defined in the previous step. You should answer these questions:

- What can users see and do before authenticating?
- What can users see and do after authenticating?
- What functions are available from the application's menus?

- Fundamentally, how do users *use* the application?

Implement

Finally, after developing a good design for your application, can begin the implementation phase.

Narrative

This section gives you a high-level overview of how a Direct to Java Client application works. It's provided as an index of programming concerns. It identifies some common issues and questions developers have when building Java Client applications and provides links to sections within this document that provide answers.

When a Direct to Java Client application launches, it asks the rule system for the specifications and actions available to the application. Specifically, it asks for the default specifications as these are the first things displayed in the user interface. You can change the available and default specifications and actions by writing rules, as described in "The Documents Menu" (page 177) and "The Default Query Window" (page 178).

The default Direct to Java Client application provides a query window for the application's main entities immediately after the application launches. A common design pattern is to provide a login window instead. A login window implementation is discussed in "Building a Login Window" (page 251) and examples are provided in both the JCDiscussionBoard and JCRealEstatePhotos examples.

The rule system determines what the application's main entities are using the heuristics described in "Entities Pane" (page 85). Rather than change how the rule system interprets an entity using Assistant, you should change your data model so that the rule system interprets that entity as the type of entity you want.

Direct to Java Client query windows have one particularly noticeable limitation: They provide query fields only for an entity's to-one relationships. So, if you want to allow users to query on an entity's to-many relationships, you'll have to provide a custom query controller.

The default application produced by Direct to Java Client is probably not exactly what you want. You can use the Direct to Java Client Assistant to easily customize the application. It allows you to choose which properties are displayed in the application's user interfaces, how those properties are aligned and placed, what types of associations the properties map to, and more. Assistant is described in "Inside Assistant" (page 85).

Assistant customizes applications by writing rules to the application's `user.d2wmodel` file. It owns this file and you should never edit it. You can add custom rules to a file named `d2w.d2wmodel` that you add to projects. Assistant won't touch the rules in that file. You should perform as many customizations in Assistant as you can because the more advanced customization techniques are more difficult, bug-prone, and are less maintainable than just using Assistant.

Although Assistant provides a user interface to configure a good number of the possible rules in a Direct to Java Client application, it is not exhaustive. Some rules that it does not provide editing of are described in "Common Rules" (page 193). You can add these rules to an application's `d2w.d2wmodel`.

After performing initial customizations of the user interface using Assistant, you want to write your application's business logic. The more advanced customization techniques make it more difficult to adapt your application to changes in your data model and business logic, so you should try to lock both down at this point. See "Business Logic Partitioning" (page 93) and "Prepare Application for Business Logic" (page 110).

After you've used Assistant to customize the client application, you can consider other customization techniques. You can freeze XML, as described in "Freezing XML User Interfaces" (page 199), which allows you to manually edit the XML hierarchy for a particular user interface in an application. You can build custom interfaces using Interface Builder or by building custom controller classes, as described in "Nondirect Java Client Development" (page 137) and "Building Custom Controllers With XML" (page 155). You may also want to consider using nib files within dynamically generated user interfaces, as described in "Mixing Static and Dynamic User Interfaces" (page 209). You can also explicitly ask the controller factory to generate user interfaces, as described in "Mixing Static and Dynamic User Interfaces" (page 209).

Certain kinds of controllers, such as those that display HTML or controllers that use the controller factory, can also be used. See "Using HTML on the Client" (page 245) and "Generating Controllers With the Controller Factory" (page 181). If you've built Cocoa applications in Interface Builder, you may find that some familiar user interface elements don't translate when you save Java Client nib files. See "Using and Extending Image Views in Nib Files" (page 221), "Using Pop-up Menus in Nib Files" (page 227), and "Using Custom Views in Nib Files" (page 213).

If you want to add custom actions to dynamically generated user interfaces, such as form windows and query windows, you add subclasses of those controller classes (subclasses of EOFormController and EOQueryController, for example) to your project, add the actions in those classes, and tell the rule system to use them for particular tasks and entities. This is described in "Extend a Controller Class" (page 130) and "Adding Custom Actions to Controllers" (page 189).

If you want to disable certain actions in dynamically generated user interfaces, you can just tell the rule system, as described in "Restricting Access to an Application" (page 177).

To add custom menu items to an application, you use D2WComponents and XML descriptions as described in "Adding Custom Menu Items" (page 185). You cannot use Interface Builder to add custom menu items to any type or part of a nondirect Java Client application.

Near the end of the development cycle, you need to think about localization. Depending on how much of the dynamism you preserve when customizing Direct to Java Client applications, localization efforts may require no more time than translating the strings. See "Localizing Dynamic Components" (page 235). When you're ready to deploy the client application, see "Deploying Client Applications" (page 169) to learn about your options.

If your customers require strict client-server security, you need to learn more about the distribution layer, as described in "The Distribution Layer" (page 93). You probably want to consider using SSL as the distribution channel ("Using SSL" (page 99)) and you'll likely implement some delegates in the distribution layer ("Delegates" (page 100)).

Java Client Concepts

This chapter introduces you to the fundamental concepts of Java Client. It defines the Enterprise Object technology and explains how it maps your database schema into Java objects. It covers Java Client architecture and includes information on the different framework layers and the functionality they provide. "Building a Simple Application" (page 47) links the concepts presented here to practical use in a sample application.

Enterprise Objects

To understand the Java Client architecture, you must first understand enterprise objects. Like all WebObjects applications, Java Client applications gain much of their usefulness by interacting with a persistent data store, usually a database. In WebObjects, databases are represented as collections of objects called enterprise objects that contain your application's business logic.

The Enterprise Object technology maps your data to these enterprise objects, and you work with the objects rather than directly with the data store. The Enterprise Object technology handles all communication with the database, which frees you from writing SQL and other database-specific code.

The Enterprise Object technology is composed of specialized layers:

- `com.webobjects.eoaccess.EOAdaptor` subclasses use JDBC or JNDI to read and write from data stores.
- `com.webobjects.eoaccess` manages interaction with a database; it is responsible for object-relational mapping.
- `com.webobjects.eocontrol` manages a graph of enterprise objects; tracks insertions into, deletions from, and changes within the object graph.
- `com.webobjects.eointerface` mediates between the control layer and an application's user interface; maps data to user interface elements.
- `com.webobjects.eodistribution` and `com.webobjects.eodistribution.client` distribute enterprise objects across the network to the client; they provide much of the functionality of the EOAccess layer on the client.
- `com.webobjects.eoapplication` and `com.webobjects.eoapplication.client` form a general user-interface utility layer specific to both types of Java Client applications.
- `com.webobjects.eogeneration` and `com.webobjects.eogeneration.rules` dynamically generate the user interface for Direct to Java Client applications.

These layers are described in more detail later in this chapter.

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 Enterprise Object technology to give and receive values for its properties.

An enterprise object is made up of its class definition (such as `com.webobjects.eocontrol.EOGenericRecord`) and the data values from the database row or record with which the object is instantiated. An enterprise object has a corresponding model that defines the mapping between the class's object model and the database schema. However, an enterprise object doesn't explicitly "know" about its model. Rather, it accesses its model through a `com.webobjects.eocontrol.EOClassDescription` object.

Enterprise Object Models

One of the fundamental features of the Enterprise Object technology is that it maps the data in data stores (usually relational databases) to objects. The industry term for this is object-relational mapping. The correspondence between an enterprise object class and stored data is established and maintained by a model. A model defines the mapping between enterprise object classes and a data store in entity-relationship terms.

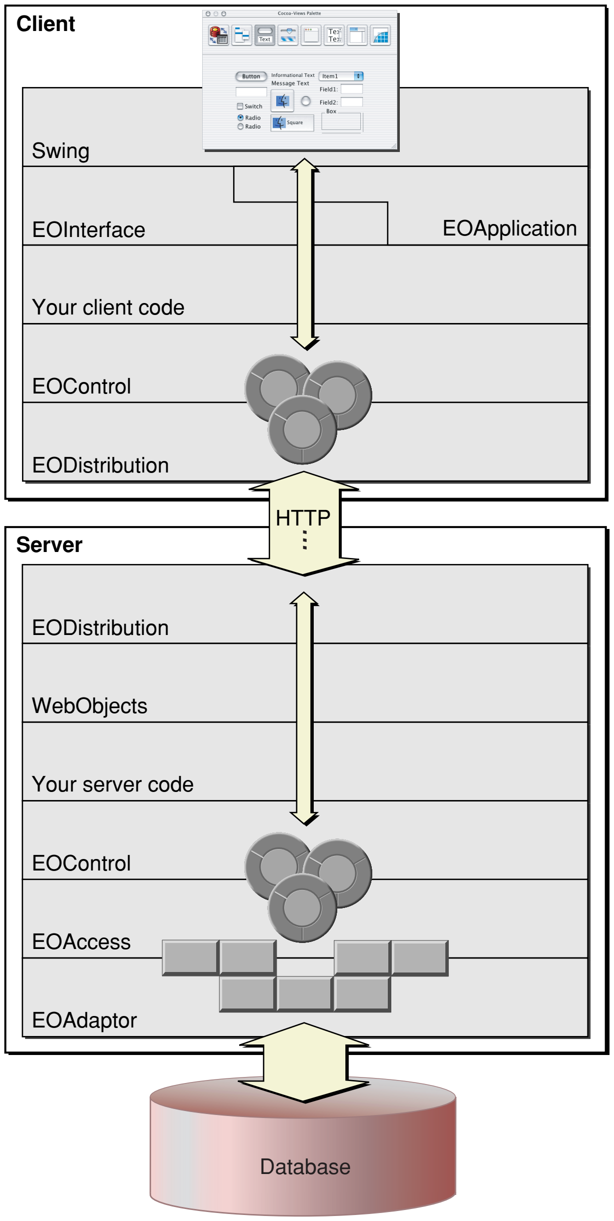
In addition to storing a mapping between the data store schema and enterprise objects, a model file stores information needed to connect to the data store. This connection information includes the name of an adaptor to load so that enterprise objects can communicate with the data store. WebObjects provides a JDBC adaptor that allows you to connect to any JDBC Type 2 compliant or Type 4 compliant database. It also provides a JNDI adaptor, and you can write your own adaptors to connect to other types of data stores.

Java Client Architecture

A Java Client application is essentially an Enterprise Objects application distributed across an application server and one or more client applications or applets.

The design of Java Client breaks up some of the layers of the Enterprise Object technology and distributes them across the client and the application server. Figure 2-1 illustrates this architecture.

Figure 2-1 Java Client architecture



The packages `com.webobjects.foundation`, `com.webobjects.eocontrol`, and `com.webobjects.eodistribution.client` are provided on the client to allow real, full-fledged, first-class enterprise objects to exist on the client side. Other technologies similar to Java Client usually implement client stubs on the client side, instead of creating real objects.

The client stub design requires a round trip to the server anytime the user does anything with the business logic on the client. In the Java Client architecture, the business logic (represented in real objects) can be queried and otherwise manipulated without making a round trip to the server. Only when the user explicitly executes a database action, such as saving or fetching, is a round trip to the server made. This is made possible because the distribution layer uses a by-copy distribution mechanism, which is described in more detail in "Java Client and Other Multitier Systems" (page 27).

Business Logic

The Enterprise Object architecture abstracts business logic from data stores and from specific data-access mechanisms. This abstraction lets you build reusable business objects that are independent of any data store or of the mechanisms for accessing data. If you build well-behaving business objects, you can easily change the data store your model accesses.

To achieve the goal of reusability, the Enterprise Object technology requires that your business logic contains no data store schema information. Business objects should not be identifiable as relating to any specific data store except by the data they contain. That is, your business objects shouldn't have any knowledge of database primary and foreign keys, JDBC code, or data store connection dictionary information. This allows you to use identical business logic classes on the client and on the server.

In Java Client applications, you must take extra control of your business logic and business objects. Unlike with HTML-based WebObjects applications, Java Client applications pass Java business logic classes (business objects) across the network. Clearly, you want to control which business logic and data each business object contains.

For instance, the client should hardly ever need to know credit card information, user passwords, algorithms specific to your business, or other sensitive business logic. Java Client defines these parameters for business logic partitioning:

- Each business object can be represented by a different class on the client and on the server.
- These different classes usually contain different sets of class properties.
- The goal in business logic partitioning is to pass as little data to the client as possible.
- Since some computations require additional data, it makes sense to let certain algorithms execute on the application server, which lives closer to the data store, and to control if this data is sent to the client.

The most important aspect of business logic partitioning is finding the partitioning scheme that minimizes the amount of data transferred from client to server. This and other business logic partitioning issues are discussed in more programmatic terms in "Business Logic Partitioning" (page 93).

Foundation Framework

The Foundation framework (`com.webobjects.foundation`) provides a set of robust and mature core classes, including utility, collection, key-value coding, time and date, notification, and debug logging classes.

Although you may choose to use the standard Java classes such as `java.util.Vector` and `java.util.HashTable`, Foundation provides a rich set of classes that you may find more flexible and robust than the standard Java foundation classes.

For historical reasons, the inner workings of WebObjects rely almost totally on Foundation for collections and other low-level functionality. In your custom classes, you are free to use the JDK foundation classes or the WebObjects Foundation classes. However, you'll find that your custom classes will be better integrated with WebObjects if written with Foundation classes.

Listed here are classes that you may find especially useful in Foundation. Consult the Foundation API reference for complete details.

- **NSKeyValueCoding** provides arbitrary access to data in objects; a better-performing alternative to standard Java set and get methods.
- **NSLog** is the WebObjects debug logging system; it allows you to easily control debug logging for everything from SQL generation to user interface generation.
- **NSBundle** provides file system and archiving services (server-side only).
- **NSDictionary and NSArray** are common data structures used in object-relational mapping.

Access Layer

The EOAccess layer (`com.webobjects.eoaccess`) is directly responsible for communicating with the data store and for registering enterprise objects with the EOControl layer. It exists only on the server and provides these functions:

- generates SQL to fetch data from and commit data to data stores
- manages the communication chain between the data store and the control layer
- manages model files, which define the object-relational mapping between data stores and Java objects
- provides classes that represent various database elements, such as tables, relationships, stored procedures, and joins
- maps raw data to business objects

EOAccess provides an elegant way to interact programmatically with data stores in an abstract manner. It is designed to work with all different types of data stores and different data store vendors, so many of its objects are portable to different data access environments. Although EOAccess is an essential element of any WebObjects application, you rarely need to use it programmatically.

Essential EOAccess Classes

The following sections introduce important EOAccess classes. For complete details, see the EOAccess API reference.

EOAdaptor

EOAdaptor defines a server-independent interface for working with relational database systems. This class is subclassed to communicate with specific data sources. Server-specific subclasses encapsulate the behavior of a specific data source.

EOAdaptor isolates your application from any particular data source. By switching the EOAdaptor your application uses, you can change data sources without changing any source code in your application.

EODatabaseContext

EODatabaseContext class has many responsibilities, including fetching, faulting, saving, and managing transactions and channels.

EOModel

EOModels establish and maintain the correspondence between an enterprise object and stored data in entity-relationship terms. EOModels also store database connection information, including the adaptor's name.

EOUtilities

EOUtilities provides a collection of static convenience methods that make working with enterprise objects easier. The methods allow you to query editing contexts for information on the entities, objects, and relationships they manage. Convenience methods are provided that allow you to more easily work with raw SQL, if necessary.

Note: EOUtilities is not available on the client because it exists in the `com.webobjects.eoaccess`, package which is not provided on the client. Furthermore, you should be careful when using EOUtilities in server-side business logic classes as some of its methods reduce the reusability of those classes.

Control Layer

The EOControl layer (`com.webobjects.eocontrol`) exists in identical form on both the client side and the server side of Java Client applications. This layer manages the object graph (a collection of enterprise objects), implements faulting (on-demand fetching), and tracks editing changes. The object store and data source used by the client control layer communicate changes to the object graph across the channel to the server.

The control layer in Java Client applications maintains an object graph on the client and on the server, but the set of objects in each object graph may differ depending on how you partition your business logic. An object that exists in both client and server object graphs is synchronized with the help of the distribution layer.

Essential EOControl Classes

The EOControl layer is very abstract, which allows it flexibility. Its abstract nature allows EOControl objects to live independent of any persistence scheme, database, or data source. The client and server parts of a Java Client application have the exact same EOControl layer; it is the layer that plugs into EOControl that differs for the client and the server. On the server side, EOControl objects talk to the database using EOAccess; on the client side, EOControl objects talk to the server using EODistribution. The EOControl classes you will encounter in development are introduced here.

EOEnterpriseObject

An EOEnterpriseObject is a flexible representation of your business logic. EOEnterpriseObjects are conceptually abstract—they are ignorant of specific data stores and data-access mechanisms. All EOEnterpriseObjects conform to these behaviors:

- **Key-value coding** is a mechanism that allows arbitrary access to data in objects without requiring instance variables. The following are examples of key-value coding accessors: `student.valueForKey("name")student.takeValueForKey("name", "Ernest").`

- **Validation** of data is done before saving, deleting, updating, and performing other operations.
- **Relationship manipulation** provides methods to facilitate the management of objects in a relationship.
- **Faulting** provides placeholders for data, rather than fetching all data at once.

These behaviors provide convenience and flexibility for your business objects, while enhancing performance and offering important business functionality.

EOEnterpriseObject is an interface, so you never instantiate it. Rather, WebObjects provides two classes that implement EOEnterpriseObject:

- **EOCustomObject** inherits from `java.lang.Object`, implements `com.webobjects.eocontrol.EOEnterpriseObject`.
- **EOGenericRecord** inherits from EOCustomObject.

EOEditingContext

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 with one another and with your data store—it is the WebObjects change-tracking mechanism. When an enterprise object changes, the EOEditingContext sends a notification so that other parts of the application, such as the user interface, can update themselves accordingly.

The EOEditingContext also manages undo and revert and is the object through which you save changes to the database. EOEditingContext is designed to abstract these database operations from your business objects, which keeps any database-specific information from living inside your business logic.

An EOEditingContext is always associated with an instance of a parent object store. In Java Client applications, the client and server have separate editing contexts. The client-side editing context is associated with a client-side object store, `com.webobjects.eodistribution.client.EODistributedObjectStore`; the server-side editing context is associated with a server-side object store, `com.webobjects.eoaccess.EODatabaseContext`.

You can think of an EOEditingContext object as an abstract database transaction object. In WebObjects, a request to fetch data from a data store is usually done from the control layer, and fetches done from the control layer almost always happen from within an EOEditingContext. Once data is fetched into objects, an EOEditingContext manages the graph of fetched objects, tracks changes to those objects, and is the object through which you invoke data store commits.

EOFetchSpecification

Because database fetches are resource-intensive operations (expensive), you rarely ask for all the data at once. Rather, you provide criteria for the data to be fetched with an EOFetchSpecification. An EOFetchSpecification describes the objects to be retrieved using an EOQualifier (an object that restricts the selection of database rows based on a specified criterion).

EOGlobalID

To maintain database independence, EOControl provides an internal mechanism to identify objects. Other systems use database primary and foreign keys to identify objects, but these keys don't represent data (they represent locations in the data store) and so shouldn't be part of your business logic. The algorithm used to generate EOGlobalIDs is designed to guarantee completely unique identifiers.

A subclass of EOGlobalID, EOTemporaryGlobalID, identifies objects before they are committed to the data store.

EOObjectStoreCoordinator

A single Java Client application can access data from different data stores. In this case, each EOModel is usually associated with a different data store, and this added complexity requires an object to manage it. Each EOModel in an application has a corresponding EODatabaseContext object. The EOObjectStoreCoordinator sits between the client's editing contexts and the EODatabaseContext objects and isolates the editing contexts from the application's data sources.

Distribution Layer

The distribution layer (`com.webobjects.eodistribution` and `com.webobjects.eodistribution.client`) synchronizes the states of the object graphs on the client and on the application server. This layer exists in part on both the client and the server and moves business objects between the two. The distribution layer on the server fetches objects and saves changes from the database and communicates these actions to the distribution layer on the client.

The server-side distribution layer contains the EODistributionContext class. It encodes data to send to the client and decodes data it receives from the client over the distribution channel. (You can implement your own encoding and decoding schemes to improve security.) It also synchronizes the server and client object graphs by tracking the state of the server-side object graph and communicating any changes to the client. EODistributionContext also validates remote invocations originating from client objects to allow only authorized invocations.

Essential EODistribution Classes

Listed here are the classes you are most likely to deal with programmatically. For complete details, see the EODistribution API reference.

- **EODistributionChannel, EOHTTPChannel.** The distribution layer provides classes for communication between the application server and client applications. EOHTTPChannel is a subclass of EODistributionChannel and implements an HTTP channel to communicate with clients. You can subclass EODistributionChannel to use a different transport protocol such as CORBA.
- **EODistributedObjectStore.** This class mediates between the distribution layer's channel (an EODistributionChannel object) and the client's editing contexts. It sends messages to its child editing contexts from the server and sends messages to the server from its editing contexts.
- **EODistributedDataSource.** Using an EOEditingContext, objects of this class fetch, insert, and delete objects from the object store. This class implements all the functionality of EODataSource, but it exists solely on the client side.

- **EODistributionContext.** This object exists on the server and is responsible for communicating with its client-side counterpart EODistributionChannel. These two objects mediate object transfer over the network and handle remote method invocation.
- **WOJavaClientComponent.** This object sits on the server side and forwards requests from the client's EODistributionChannel to the server's EODistributionContext. It also plays a critical role in application initialization.

See "The Distribution Layer" (page 93) for more information on the distribution layer and to better understand how these objects work together.

Client Interface Layer

The EOInterface layer (`com.webobjects.eointerface`) displays to the user the properties of the enterprise objects maintained in the client control layer. 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 associations and display groups.

Display Groups

A display group (`com.webobjects.eointerface.EODisplayGroup`) coordinates the flow of data between the user interface and the database. Display groups decide what data to allow associations to display. They fetch data from either database contexts or other display groups through `com.webobjects.eocontrol.EODataSource` objects.

Associations

As mentioned earlier, associations keep the user interface synchronized with enterprise object values. Associations in Java Client derive from EOAssociation, an object that maintains a two-way binding between the properties of a display object and the properties of one or more enterprise objects contained in EODisplayGroups.

EOAssociation defines the different parameters of the display object it controls using **aspects**. These parameters include the values displayed and whether the display object is enabled or editable. Each aspect of a display object can be bound to an EODisplayGroup object with a key denoting the property of its associated enterprise object.

For instance, EOTableAssociation (`com.webobjects.eointerface.EOTableAssociation`) defines these aspects:

- `source`—the object from which the table's data is fetched, usually a display group.
- `bold`—sets a flag to make the text in the table bold.
- `italic`—sets a flag to make the text in the table italics.
- `textColor`—defines the color of the text in the table.
- `enabled`—a flag that controls editability, usually associated with an attribute in a display group.

The EOInterface framework includes associations for different types of user interface objects, such as table columns, text fields, and checkboxes. Each association has multiple aspects. Associations are defined in the EOInterface framework. See the EOInterface API reference for complete details.

Typically, you create and configure associations in Interface Builder when you build user interfaces by hand. Associations are created and configured automatically if you use the dynamic user interface generation of the Direct to Java Client approach. See the EOInterface API reference for information on configuring associations programmatically.

There are many different kinds of associations. These are some of the more common ones:

- **EOActionAssociation.** Sits between an action widget (such as a button) and a display group. Reacts to a mouse click or a keypress and invokes a particular business method, based on the bound aspect.
- **EOMasterDetailAssociation.** These associations bind one display group (the detail display group) to a relationship in another display group (the master display group) so that the detail display group contains the destination objects for the object selected in the master display group. Takes a relationship key rather than an entity name and displays a subset of data in the master display group.
- **EOTableAssociation.** Maps all the objects in a display group to a user interface table view. This association takes no direct keys, but uses an EOTableColumnAssociation, which take keys.
- **EOTextFieldAssociation.** Takes a value key that determines the property to be displayed in or taken from the text field.
- **EOValueAssociation.** Associates a single property of the value display group's selected object with a widget.

Application Layer

The EOApplication layer, defined in `com.webobjects.eoapplication` and in `com.webobjects.eoapplication.client`, isolates the developer from the idiosyncrasies of each client-side execution environment (Web Start, desktop applications, or applets). It provides the classes that are used to manage application-level data and resources, including transient and persistent defaults, localization information, menu operations like save and quit, documents, user interface controls, and so on.

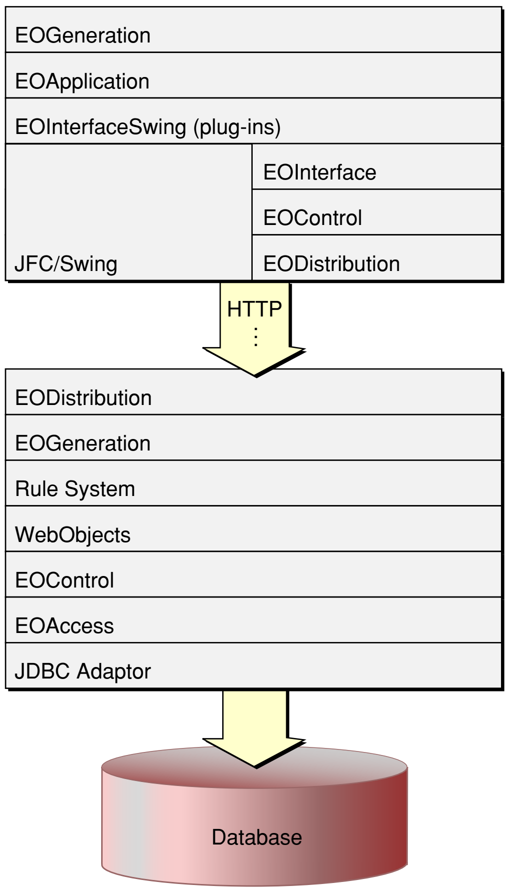
JFC/Swing does not provide a full suite of application logic utility classes, so the Java Client application layer steps in and provides other basic services as well, such as application startup and shutdown.

Generation Layer

The EOGeneration layer, defined in `com.webobjects.eogeneration` and `com.webobjects.eogeneration.rules`, dynamically generates user interfaces in Java Client applications that use the rule system. It is not used in nondirect Java Client applications. This layer analyzes your application's business model (defined in an EOModel) and, using a sophisticated set of rules, generates a user interface. The user interface description is then sent to the client where it is executed. You can alter the rules in a number of ways for customization purposes.

The generation layer consists of predefined controller classes that are dynamically mapped to Swing user interface objects and Enterprise Objects at runtime. The generation layer uses the WebObjects rule system as part of this dynamic user interface generation. The rule system and the generation layer are the elements that make a Direct to Java Client application different from a nondirect Java Client application. They are illustrated in Figure 2-2.

Figure 2-2 The complete stack of WebObjects layers in Direct to Java Client



Model-View-Controller Paradigm

A common and useful paradigm for object-oriented applications, particularly business applications, is Model-View-Controller (MVC). Derived from Smalltalk-80, MVC proposes three types of objects in an application, separated by abstract boundaries and communicating with each other across those boundaries.

Model objects represent special knowledge and expertise, such as a company's data and business logic. Model objects are not directly displayed. They often are reusable, distributed, persistent, and portable to a variety of platforms.

View objects represent things visible in the user interface such as windows, table views, and buttons. A View object is "ignorant" of the data it displays, as it relies exclusively on the Controller object for data. View objects tend to be very reusable and so provide consistency between applications.

The Controller object acts as a mediator between Model objects and View objects. Usually there is one Controller per application or per window. Controller objects communicate data back and forth between the Model objects and the View objects. A Controller's function is usually very specific to an application, so it is generally not reusable like View and Model objects are.

Because of the Controller's central mediating role, Model objects need not know about the state and events of the user interface, and View objects need not know about the programmatic interfaces of Model objects.

Within the MVC paradigm, enterprise objects are Model objects. By definition, Model objects represent data and business logic. The Enterprise Object technology extends the MVC paradigm so enterprise objects are independent of their persistent storage mechanism. Enterprise objects do not need to know about the database that holds their data, and the database doesn't need to know about the enterprise object formed from its data.

Building a Simple Application

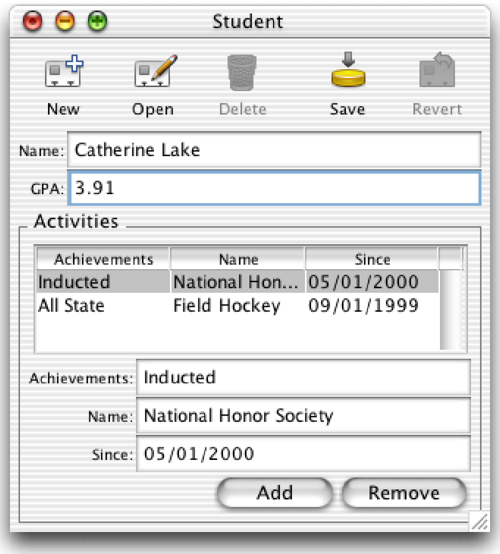
This chapter leads you through the creation of a Java Client application starting with the Direct to Java Client project type. You'll learn how to

- create a simple database using OpenBase Manager
- create tables in that database using EOModeler
- build a Direct to Java Client application using Project Builder
- perform simple customizations of the application using the Direct to Java Client Assistant

Note: Projects for the tutorials in this book are available on the WebObjects documentation home page: http://developer.apple.com/documentation/WebObjects.

You'll create a simple college admissions application with a rich user interface and database access. The application stores records of prospective students, which allows you to track students throughout the admissions process. Figure 3-1 shows a sample student record from this application.

Figure 3-1 Part of the application created in this chapter



Name: Catherine Lake
GPA: 3.91

Achievements	Name	Since
Inducted	National Hon...	05/01/2000
All State	Field Hockey	09/01/1999

Achievements: Inducted
Name: National Honor Society
Since: 05/01/2000

Create the Database

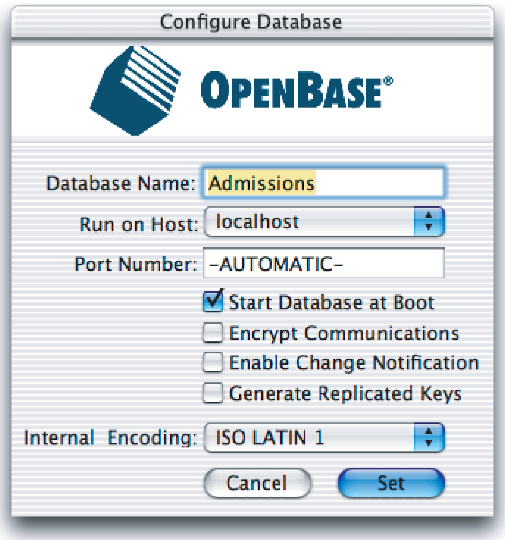
The WebObjects developer software package includes a limited-use version of OpenBase, a SQL database server. Follow these steps to configure a new OpenBase database:

1. In Mac OS X, navigate to `/Applications/OpenBase` and launch OpenBase Manager.

2. Choose New from the Database menu.

3. Name the database `Admissions`. Select the Start Database at Boot option. Choose ISO LATIN 1 from the Internal Encoding pop-up menu. The Configure Database dialog should appear as shown in Figure 3-2.

Figure 3-2 Configuring a new database

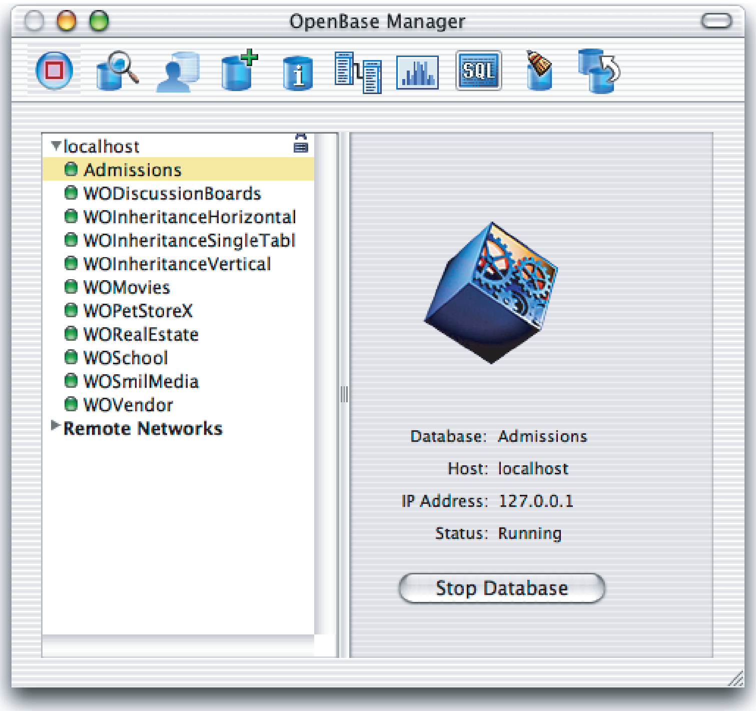


Database Name: Admissions
Run on Host: localhost
Port Number: -AUTOMATIC-
[✓] Start Database at Boot
[ ] Encrypt Communications
[ ] Enable Change Notification
[ ] Generate Replicated Keys
Internal Encoding: ISO LATIN 1

4. Click Set.

5. Select the new database in the database list under localhost and start it manually. Make sure the database is started (denoted by the green icon, as Figure 3-3 shows).

Figure 3-3 OpenBase Manager main window



Database: Admissions
Host: localhost
IP Address: 127.0.0.1
Status: Running

6. Quit OpenBase Manager.

Create an EOModel

EOModeler is a powerful application that provides tools to build a data model that describes the entity-relationship mapping between the data sources your application uses and the Java business objects that bring that stored data to life. Its product is an EOModel, which contains database connection information, such as the database adaptor, version number, and login information. EOModels also form the foundation of your business logic—they offer an object-oriented view of the tables and relationships in your database. You use EOModeler to

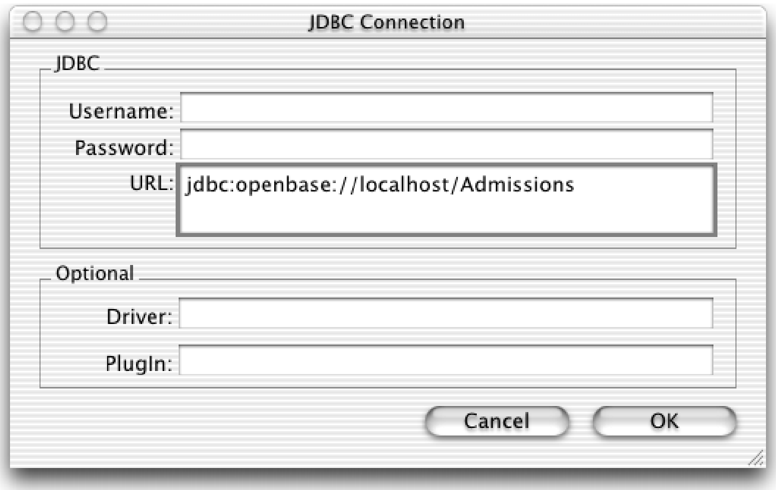
- create tables and relationships in a database
- generate SQL
- generate client and server Java files based on EOModels
- build fetch specifications

A good model is important because Direct to Java Client's dynamic user-interface generation system analyzes EOModels and generates user interfaces from them. If you build good models, Direct to Java Client can generate a full-featured application automatically without requiring you to write a line of code. In fact, a Direct to Java Client application is a great way to test the integrity of EOModels.

Follow these steps to create an EOModel:

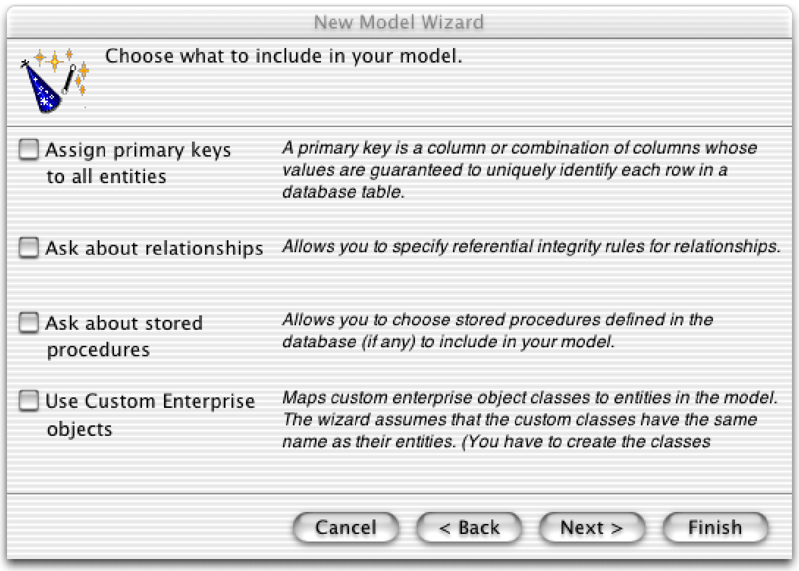
1. In Mac OS X, navigate to `/Developer/Applications` and launch EOModeler.
2. Choose New from the Model menu.
3. Select JDBC as the adaptor.
4. In the JDBC Connection window, enter `jdbc:openbase://localhost/Admissions` in the URL field, as shown in Figure 3-4 (page 50). Click OK.

Figure 3-4 JDBC connection information


JDBC Connection
JDBC
Username:
Password:
URL: jdbc:openbase://localhost/Admissions
Optional
Driver:
PlugIn:

5. Since the database is empty, deselect the four options in the next window and click Next. See Figure 3-5.

Figure 3-5 Deselect all options for this model


New Model Wizard

Choose what to include in your model.

- [ ] Assign primary keys to all entities — *A primary key is a column or combination of columns whose values are guaranteed to uniquely identify each row in a database table.*
- [ ] Ask about relationships — *Allows you to specify referential integrity rules for relationships.*
- [ ] Ask about stored procedures — *Allows you to choose stored procedures defined in the database (if any) to include in your model.*
- [ ] Use Custom Enterprise objects — *Maps custom enterprise object classes to entities in the model. The wizard assumes that the custom classes have the same name as their entities. (You have to create the classes*

6. There are currently no tables in the database, so click Finish in the Choose Tables to Include dialog.

Behind the Steps

Step 3: WebObjects 5 supports databases with JDBC Type 2 and Type 4 connectivity. Oracle, OpenBase, MSSQL 2000, Sybase, and MySQL are qualified for WebObjects 5.2. See the document *Post-Installation Guide* for more exact specifications. Third parties have developed JDBC adaptor plug-ins for other JDBC-compliant databases. See the Apple Support Knowledge Base for information on creating custom JDBC adaptor plug-ins.

WebObjects database connectivity is not limited to JDBC-compliant databases. In principle, you can also write adaptors for ERP systems and even flat file systems. WebObjects 5.2 also supports data stores with JNDI connectivity.

Step 5: EOModeler works by reverse-engineering your database. So, if your database is already populated with tables, primary keys, relationships, and stored procedures, you can tell EOModeler to consider these attributes when building a model.

- Assign primary keys to all entities—When reading and writing to databases, the access layer of Enterprise Objects uses primary keys to uniquely identify enterprise objects and to map them to the appropriate database row. Therefore, each entity in your model needs a primary key. The EOModeler Wizard automatically assigns primary keys to the model if it finds primary key information in the database.

  However, if primary keys aren't defined in the database schema information, the wizard prompts you to choose primary keys.

Note: Although Enterprise Objects uses primary keys when reading and writing to the database, it assigns an identifier (an EOGlobalID) to each enterprise object. This allows business logic to be independent of database primary and foreign keys, which makes your business objects reusable. To reiterate, although Enterprise Objects needs to know about database primary keys, your business logic should never explicitly reference database primary or foreign keys.

- Ask about relationships—If the Wizard finds foreign key definitions in the database schema information, it includes the corresponding relationships in the model. However, foreign key definitions in the schema don't provide enough information for the Wizard to set all of a relationship's options. If you select this option you will be prompted to provide additional information, such as the join type, delete rule, batch faulting batch size, and more.
- Ask about stored procedures—Selecting this option causes the Wizard to display the stored procedures it finds in the schema and allows you to choose which to include in your model.
- Use Custom Enterprise objects—Each entity in the model corresponds to a table in the database and each has a corresponding Java class. This Java class can be an instance of `com.webobjects.eocontrol.EOGenericRecord` or a custom subclass of EOGenericRecord.

  If you deselect this option, the Wizard maps all database tables to EOGenericRecord classes. Otherwise, it maps each entity to a subclass of EOGenericRecord of the same name (a table named "STUDENT" corresponds to an entity named "Student," which corresponds to a Java class named "Student.java.")

  You use custom enterprise object classes to add custom business logic to your application (which is quite common).

Build the Model

EOModeler creates an empty model containing just a database connection dictionary, which specifies the adaptor type, database URL, and other basic information. Click the root of the object tree (probably titled "UNTITLED0"), and then choose Inspector from the Tools menu to see the database connection dictionary.

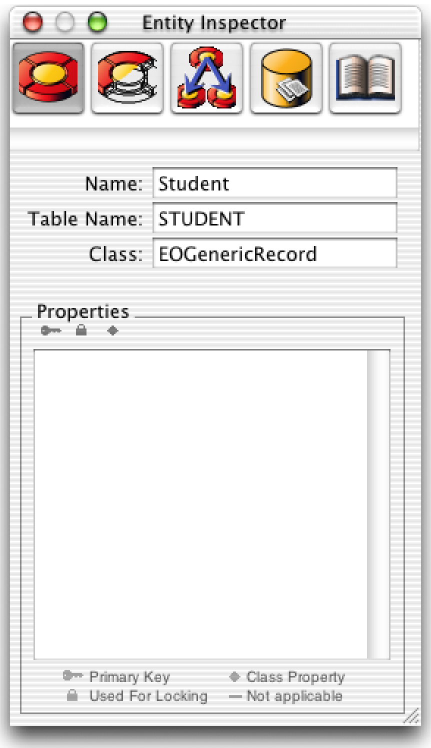
Note: The model you create will initially be suboptimal so that the tutorial can demonstrate some features of Java Client you wouldn't otherwise see with a perfect model.

Follow these steps to add a table with attributes to the model:

1. Create a new entity by selecting Add Entity from the Property menu.
2. Select Inspector from the Tools menu if it is not already present. When you created an entity in step 1, the Inspector focused on that entity, so its title is now "Entity Inspector."

3. In the Entity Inspector, change the Name field to `Student` and the Table Name field to `STUDENT`. Leave the Class field EOGenericRecord. See Figure 3-6.

Figure 3-6 Entity Inspector



Name: Student
Table Name: STUDENT
Class: EOGenericRecord

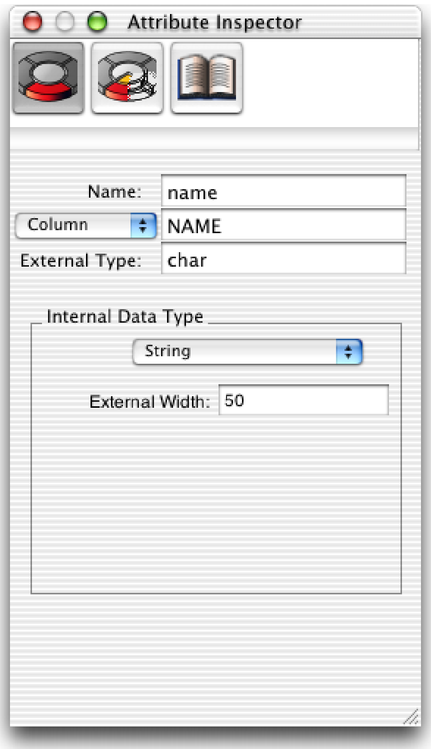
4. Select the Student entity in the tree view and add a new attribute to it by selecting Add Attribute from the Property menu. The title of the Inspector window changes to "Attribute Inspector."

5. In the Attribute Inspector, change the Name field to `name` and the Column name to `NAME`.

6. In the External Type field, enter `char`.

7. Choose String from the Internal Data Type pop-up menu, and enter `50` in the External Width field, as shown in Figure 3-7. Make sure to press Enter or tab out of the field so the value sticks.

Figure 3-7 Attribute Inspector for the `name` attribute



Name: name
Column: NAME
External Type: char
Internal Data Type: String
External Width: 50

8. Add a second attribute named `gpa` with Column name `GPA`. Enter `int` in the External Type field and choose Integer for Internal Data Type. The types selected here are the suboptimal part of the model that will be corrected in a later step.

Behind the Steps

Step 3: In this book, the naming conventions for entities and attributes follow standard Java naming conventions and common relational database conventions.

Entities adhere to the naming convention for Java classes: The name begins with a capital letter, and the first letter of inner words is capitalized, such as "NewStudent."

Table names adhere to the common relational database convention of capitalizing every letter, and separating inner words with the underscore (_) character, such as "NEW_STUDENT."

Attribute names follow the Java convention for methods: The name begins with a lowercase letter, and the first letter of inner words is capitalized, such as "firstName."

Column names adhere to the same database conventions that tables do.

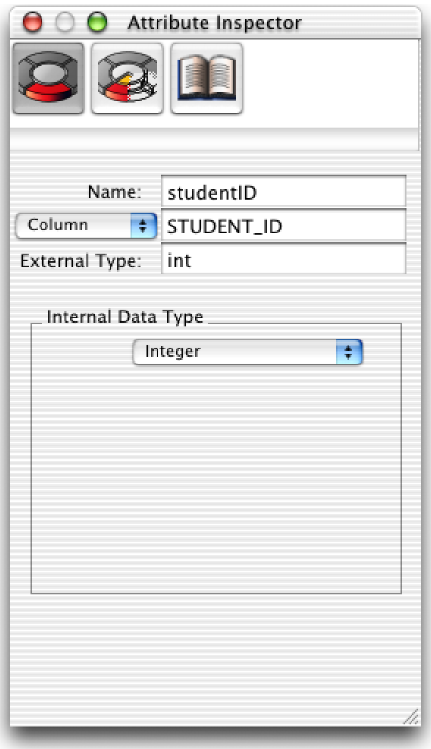
Step 6: When adding attributes, you can choose the external type from a pop-up menu in EOModeler's table view, rather than type it in. Simply click the downward pointing arrow to the right of a row in the External Type column. Doing this will also familiarize you with the different external types for the database you are using.

Completing the Model

Simply creating entities with attributes does not make a complete model. You must also assign a primary key to the entity and select certain properties to send to the client. Follow these steps to complete the basic model:

1. Add a third attribute to the Student entity named `studentID`. The column name is `STUDENT_ID`. Give it an external type of `int` and choose Integer for the internal data type. This attribute is the entity's primary key. See Figure 3-8.

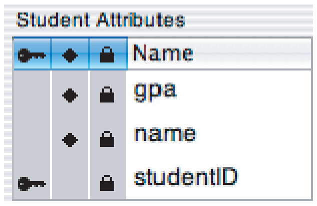
Figure 3-8 The primary key attribute



Name: studentID
Column: STUDENT_ID
External Type: int
Internal Data Type: Integer

2. In table mode (Tools > Table Mode), there are three icons in the table heads of the attributes pane for an entity, as shown in Figure 3-9.

Figure 3-9 Default attribute columns in table mode


Student Attributes

🔑	◆	🔒	Name
	◆	🔒	gpa
	◆	🔒	name
🔑		🔒	studentID

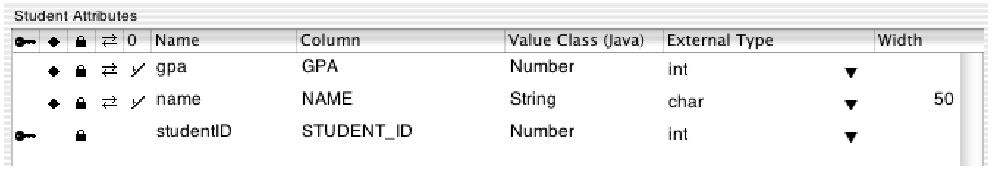
If the key icon is present for a particular attribute, that attribute is or is part of the entity's primary key. If the diamond icon is present for a particular attribute, that attribute is a server-side class property. If the pad lock icon is present for a particular attribute, that attribute is used for optimistic locking. Make the `studentID` attribute the primary key by clicking in its key field.

3. Unmark the primary key (`studentID`) as a server-side class property by clicking the diamond icon to its left.

4. To select which attributes are available to the client-side application, you need to add a view column in table mode. From the Add Column pop-up menu, choose Client-Side Class Property. This adds a column with two opposing arrows to the attribute's table. (You can remove columns by selecting a column and pressing the Delete key). Make sure that only the `gpa` and `name` attributes are selected as client-side class properties, as shown in Figure 3-10 (page 56).

5. Save the model as `Admissions.eomodeld`. It should look like Figure 3-10 (page 56), though the columns you see in the table view may be different if you've added or removed columns from it.

Figure 3-10 The finished model


Student Attributes

🔑	◆	🔒	⇄	0	Name	Column	Value Class (Java)	External Type	Width
	◆	🔒	⇄	✓	gpa	GPA	Number	int ▼	
	◆	🔒	⇄	✓	name	NAME	String	char ▼	50
🔑		🔒			studentID	STUDENT_ID	Number	int ▼	

Behind the Steps

Step 2: Each of the records in a table must be unique—no two records can contain exactly the same values. To ensure this, each entity must contain an attribute that's guaranteed to represent a unique value for each record; this value is called the entity's primary key.

By default, EOModeler makes all of an entity's attributes class properties. 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.

There are two types of class properties: client-side class properties and server-side class properties. EOModeler indicates that an attribute is a server-side class property with the diamond icon. Client-side class properties are represented by the double-arrow icon.

Step 3: Primary keys are of no use to client-side classes, so they need to be unmarked as client-side class properties.

Step 4: Likewise, primary keys are of no use to server-side classes, so they need to be unmarked as server-side class properties.

Generate SQL

Now that you've built an EOModel, you need to write the table information to the database. Fortunately, EOModeler generates SQL for you; just follow these steps:

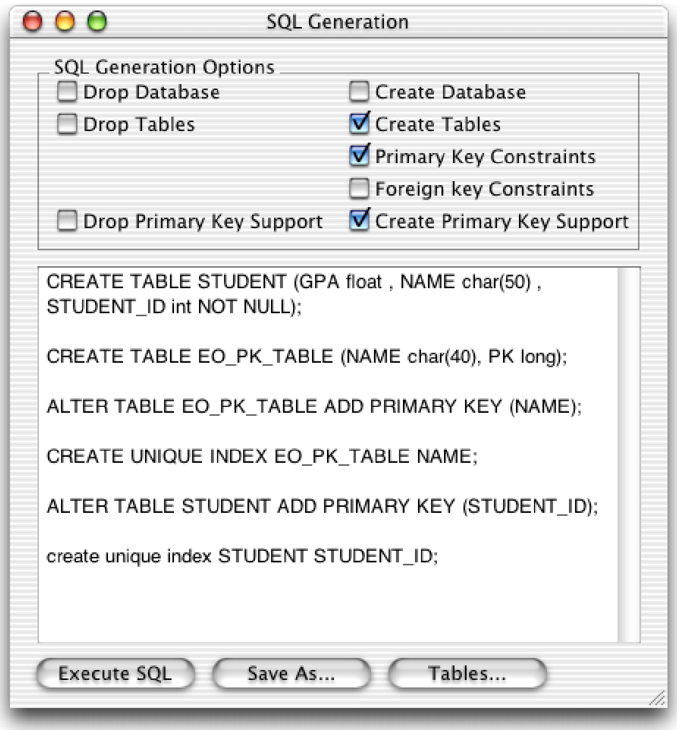
1. Select the Student entity in the tree view.

2. Choose Generate SQL from the Property menu.

3. Deselect all options except Create Tables, Primary Key Constraints, and Create Primary Key Support, as shown in Figure 3-11.

4. Click Execute SQL. The SQL generated is specific to OpenBase. EOModeler knows to generate OpenBase-specific SQL because of the adaptor plug-in the model is using. Were the model connecting to an Oracle database, Oracle-specific SQL would instead be generated.

Figure 3-11 Generate SQL



- [ ] Drop Database
- [ ] Drop Tables
- [ ] Drop Primary Key Support
- [ ] Create Database
- [✓] Create Tables
- [✓] Primary Key Constraints
- [ ] Foreign key Constraints
- [✓] Create Primary Key Support

```
CREATE TABLE STUDENT (GPA float , NAME char(50) ,
STUDENT_ID int NOT NULL);

CREATE TABLE EO_PK_TABLE (NAME char(40), PK long);

ALTER TABLE EO_PK_TABLE ADD PRIMARY KEY (NAME);

CREATE UNIQUE INDEX EO_PK_TABLE NAME;

ALTER TABLE STUDENT ADD PRIMARY KEY (STUDENT_ID);

create unique index STUDENT STUDENT_ID;
```

5. To verify the table was written to the database, in OpenBase Manager, select the Admissions database and then choose Schema from the Database menu. You should see two tables: EO_PK_TABLE and STUDENT. Select the Student table and verify that the attributes you added to the model were written to the database.

Behind the Steps

Step 3: EOModeler's SQL generation feature generates database-specific SQL based on the EOAdaptor chosen for the model. These are the eight SQL generation options:

- **Drop Database** deletes all entity tables, key constraints, and primary key support tables. This option may not be available for some data stores.
- **Drop Tables** deletes only the entity tables selected in EOModeler's main window.
- **Drop Primary Key Support** deletes primary key support from the database; for OpenBase databases, this option deletes the EO_PK_TABLE table.
- **Create Database** generates tables in the database for all entities in the EOModel.
- **Create Tables** generates tables in the database only for the models selected in EOModeler's main window.

- **Primary Key Constraints** generates database-specific key constraints.
- **Foreign Key Constraints** generates database-specific key constraints.
- **Create Primary Key Support** generates the `EO_PK_TABLE` for OpenBase databases.

Step 5: The EO_PK_TABLE table is used by Enterprise Objects when it generates primary keys. This table is written only to certain data sources when you generate SQL from EOModeler. The table is not generated, for example, when the data source is an Oracle database.

Create the Project

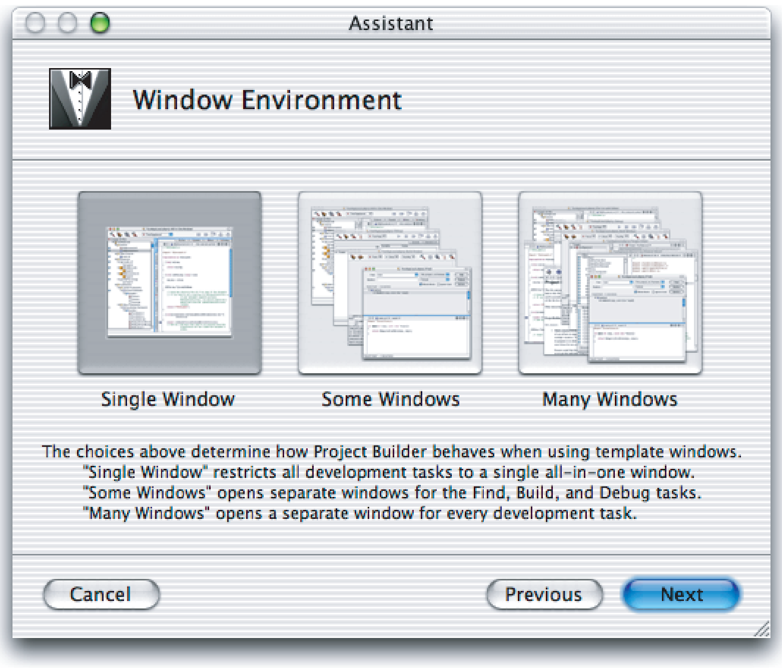
Project Builder is the WebObjects integrated development environment. Its many functions include these:

- creating working projects from project templates
- organizing project files and resources
- source-code editing
- compiling and debugging projects
- running projects
- communicating with other WebObjects development tools

Project Builder is a complex, feature-rich development environment that is designed to support many types of application development from Mac OS X device drivers to WebObjects rule-based Java applications. Because it supports so many different kinds of development, it takes some time to learn.

When you launch Project Builder for the first time, you'll be asked a few questions including what kind of user interface you want to use. The Window Environment pane of the Project Builder setup assistant appears as shown in Figure 3-12.

Figure 3-12 Project Builder's Window Environment options



The choices above determine how Project Builder behaves when using template windows.
"Single Window" restricts all development tasks to a single all-in-one window.
"Some Windows" opens separate windows for the Find, Build, and Debug tasks.
"Many Windows" opens a separate window for every development task.

The first versions of Project Builder in Mac OS X limited you to a single window in which all files and operations in a project occurred. With the Mac OS X 10.2 version of Project Builder, a new multiwindow user interface is available, which provides a much more flexible workspace.

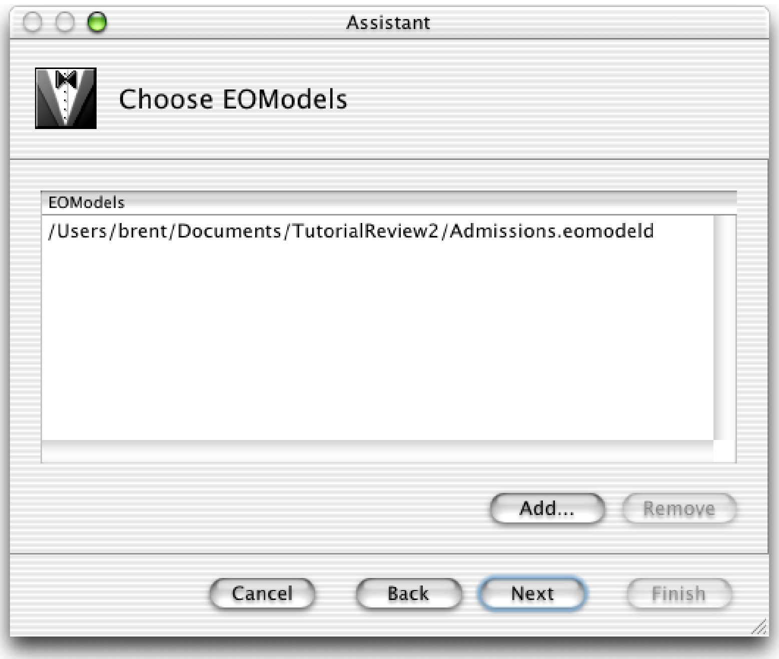
The tutorials in this book assume you're working in Single Window mode but you're free to choose another window mode if you're comfortable with Project Builder.

Project Builder provides an assistant to help you build a Java Client application starting with the Direct to Java Client project type. Follow these steps to create a new project:

1. In Mac OS X, navigate to `/Developer/Applications` and launch Project Builder.
2. Choose New Project from the File menu.
3. Select Direct to Java Client Application (Three Tier) under the WebObjects group as the new project type and click Next.
4. Name the project `Admissions` and choose a location in the file system that has no spaces in the complete pathname. Click Next.
5. In the Enable J2EE Integration pane, make sure neither option is selected, and click Next.
6. In the Choose Adaptors pane, select JavaJDBCAdaptor.framework and click Next.
7. In the Choose Frameworks pane, click Next.

8. In the Choose EOModels pane, click Add and select the EOModel you just created. Then click Next. See Figure 3-13.

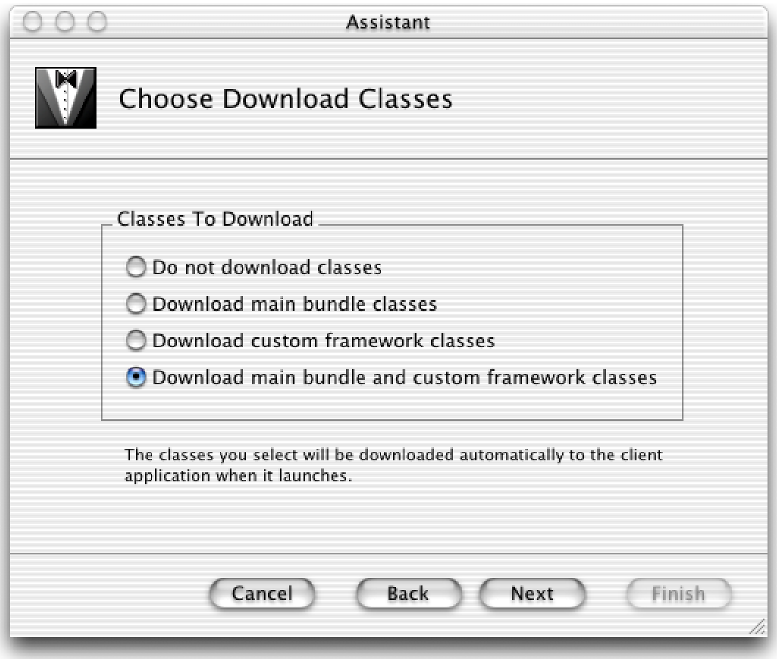
Figure 3-13 Choose the EOModel



/Users/brent/Documents/TutorialReview2/Admissions.eomodeld

9. In the Choose Download Classes pane, select the option "Download main bundle and custom framework classes" and click Next. See Figure 3-14.

Figure 3-14 Configure the class loader



Classes To Download

( ) Do not download classes
( ) Download main bundle classes
( ) Download custom framework classes
[✓ Download main bundle and custom framework classes]

The classes you select will be downloaded automatically to the client application when it launches.

10. In the Web Start pane, you can enter your company's name in the Vendor Name field and a description of the application in the Description field, but neither is required. This pane is used to configure the Web Start JNLP file when deploying the client application.

11. In the Build and Launch Project pane, make sure "Build and launch project now" is selected and click Finish. Project Builder sets up the project, builds it, and runs it. If you're developing in Mac OS X, the client application is automatically launched. If you're developing in Windows, however, you must manually launch the client application. See "Add a Launch Argument" (page 67) to learn how to manually build and run the application.

    If you're developing in Mac OS X, you can skip to "Using the Application" (page 71). Or, if you want to learn more about the default project, Project Builder, launch arguments, and manually running the client application, continue with the next section.

Behind the Steps

Step 7: It is common to build a custom framework to contain your EOModels and other custom business logic. You add custom frameworks to your project in this step.

Step 8: The EOModel you select is copied into your project's directory. From this point on, open the model from within the project to edit it.

Step 9: This step configures the Java Client Class Loader feature that first shipped with WebObjects 5.1. It facilitates the download of classes to the client for Java Client applications that are deployed as desktop applications. It does not apply to client applications that are deployed with Web Start or as applets. The class loader has four configuration options:

- **Do not download classes** suppresses the class loader.
- **Download main bundle class** downloads the `.woa` build product that includes custom Java classes defined in your project (but not classes defined in custom frameworks).
- **Download custom framework classes** downloads custom frameworks that your project links against, including custom Java classes in these frameworks.
- **Download main bundle and custom framework classes** downloads the `.woa` build product and custom frameworks your project links against.

More About the Java Client Class Loader

Unlike applets running in browsers, Java desktop applications do not have an automatic mechanism to download classes. This usually requires you to install the complete application manually, which can be inconvenient and makes updating the software complicated.

But with the new Java Client Class Loader feature you need only install a Java Client base system (including Foundation, EOControl, and EOAccess) on the client and download all classes specific to your application (business logic, interface controllers, user interface code, and so forth) at startup time. You configure whether and which classes should be downloaded through bindings of the WOJavaClientComponent of the WebObjects server-side application (the Project Builder Assistant for Java Client projects configures these bindings for you based on the selection you make in the Choose Download Classes pane). All you have to supply on the client is the base Enterprise Objects stack, which is contained in the `wojavaclient.jar` file.

The four possible bindings for the Java Client Class Loader are

- `noDownloadClientClasses`
- `mainBundleClientClasses`
- `customFrameworksClientClasses`
- `customBundlesClientClasses`

This feature is useful for deployment (since installing an update of the client desktop application is necessary only when you switch the version of WebObjects you use, not when you update your own custom classes) and for development (since you can create generic launch programs or scripts without worrying about the classpath).

The Default Project

For Direct to Java Client projects, the Project Builder Assistant creates a fully functional application. Take a moment to examine the default project.

As in all WebObjects applications, `Application.java`, `Session.java`, `DirectAction.java`, and `Main.java` are present, along with `Main.wo`. In the Resources group, notice that there is no interface file (no nib file), only the EOModel and an empty Direct to Web model (the `user.d2wmodel` file) to store rules generated by the Direct to Java Client Assistant.

Project Builder offers several tools that allow you to visually organize all the files in a project. 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 other groups under general categories. There is no restriction on the type of file you can put in a group.

When you create a Direct to Java Client application, Project Builder creates a default hierarchy with eight major groups. You can modify this organization by adding, removing, or deleting groups, and by moving files between groups. Keep in mind that groups are useful only for organizational purposes: they have no effect on how their content or the application behaves.

These are the eight major groups in a Direct to Java Client application:

- **Classes** stores the core Java files (`.java`) in the project such as `Application.java`, `Session.java`, and `DirectAction.java`. The Java files related to components, such as `Main.java`, are by default organized in subgroups of the Web Components group.
- **Web Components** stores the WebObjects components used in your project. By default, Java Client applications have a single WebObjects component, `Main.wo`. Later on, you'll put frozen XML user-interface components in this group.
- **Resources** stores the model files (`.eomodeld`) used in the project as well as custom rule files (`d2w.d2wmodel`) and Direct to Java Client Assistant's `user.d2wmodel` file.
- **Web Server Resources** contains image files (`.gif`, `.jpg`, `.png`) and localizable string tables.
- **Interfaces** contains Interface Builder files (`.nib`) for Java Client applications or for Direct to Java Client applications using frozen interface files.
- **Frameworks** is a visual representation of the frameworks your project links against at compile and runtime.
- **Documentation** contains documentation for your application.
- **Products** contains the build application as well as intermediate build files.

You can freely move files to different groups, rename groups, and remove groups. The only attribute of a project file that really matters is the target with which each file is associated.

Targets

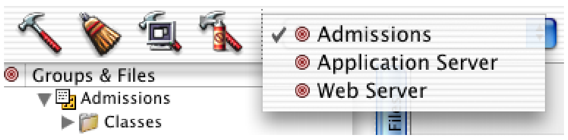
When built, Java Client applications include two products: the client product and the server product. The client product is the client-side application and the server product is the server-side application. The client product is the result of the files built for the Web Server target. The server product is the result of the files built for the Application Server target.

The Web Server and Application Server targets are build targets and the Admissions target (or the target named after your application) is the root or aggregate target.

- **Build targets** are used to configure the settings for a particular target, either the client application or the server application. When you define a build target, you tell Project Builder which files are a part of the target and how to build the target's product.
- **Root targets** or aggregate targets are used to group two or more build targets into a single unit. No files are associated with root targets except through their association with build targets. When an aggregate target is built, the build targets it contains are built in turn. The root target is the target you compile on.

Use the Target pop-up menu to switch between a project's targets, as Figure 3-15 (page 65) shows.

Figure 3-15 Target pop-up menu



Client Files (Web Server Target)

For Java Client applications, the files associated with the Web Server target are Interface Builder archive files (`.nib`), interface controller classes (`.java`), custom controller classes (`.java`), client-side image resources (`.gif`, `.jpg`, `.png`), client-side business logic classes (`.java`), and client-side localized string tables (`Localizable.strings`).

Server Files (Application Server Target)

The server-side project files created by Project Builder are distributed across several groups. Most notable of these are the two WebObjects components (WOComponent classes). Starting in WebObjects 5.2, Java Client projects include both a Main component (`Main.wo`) and a JavaClient (`JavaClient.wo`) component. Both components are put in the Web Components group by default. The two components work together to provide the client application via Web Start.

The `Main.html` file in `Main.wo` contains this markup:

```
<HTML>
    <HEAD>
        <TITLE>Main</TITLE>
    </HEAD>
    <BODY>
        <CENTER>
            Please
            <WEBOBJECT NAME=JavaClientLink>click here</WEBOBJECT>
            to start RealEstatePhotos through Web Start.
        </CENTER>
    </BODY>
</HTML>
```

The `Main.wod` file contains this code:

```
JavaClientLink: WOHyperlink {
    href = javaClientLink;
}
```

The `<WEBOBJECT NAME=JavaClientLink>` tag in `Main.html` is bound to the definition of `JavaClientLink` in `Main.wod`, which resolves to the value of the `javaClientLink` method in the component's class file (`Main.java`). That method returns a URL that points to the Web Start JNLP file of the client application.

The other WOComponent class, `JavaClient.wo`, stores information that is used to dynamically generate the Web Start JNLP file when a user clicks the hyperlink in `Main.wo` that resolves `<WEBOBJECT NAME=JavaClientLink>`. This component contains a number of interesting bindings.

The most important of these is the `applicationClassName` binding. This binding is the switch that determines if a Java Client application is of the direct type or nondirect type. As the project type in this tutorial is of the direct type, the binding specifies `com.webobjects.eogeneration.EODynamicApplication`. The default binding is `com.webobjects.eoapplication.EOApplication`, so if the binding is not present in `Main.wod`, the default is assumed (this is the case for projects begun with the nondirect project type).

The `applicationName`, `applicationDescription`, and `vendor` bindings are used in the Web Start JNLP file, and the `downloadClientClasses` binding is used to configure the Java Client Class Loader feature (see "More About the Java Client Class Loader" (page 63) for more details). The optional `interfaceControllerClassName` binding specifies the fully qualified class name of a `.nib` file to load at application launch time.

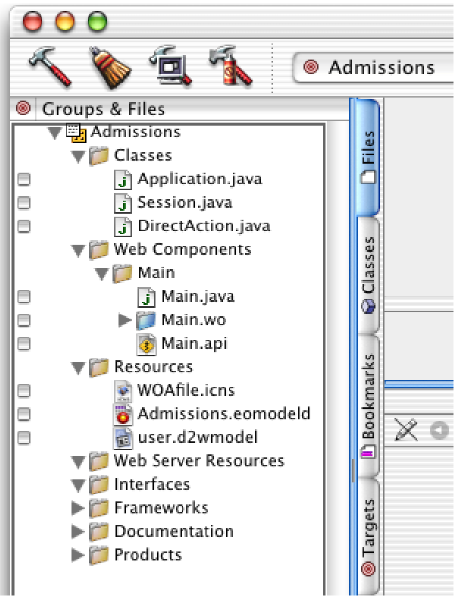
Bindings are also available to customize the behavior of the distribution layer. See "Distribution Layer Objects" (page 98) for more information on these bindings.

Other server files include

- the `Application.java`, `Session.java`, and `DirectAction.java` class files
- any EOModels your application uses
- the exported bindings for the Main component (`Main.api`)

Figure 3-16 shows the default groups and files.

Figure 3-16 The default groups and files



The next section continues building the tutorial project.

Add a Launch Argument

Java Client applications have usage patterns that are more like those of desktop applications rather than those of HTML applications. Desktop applications are often left open for hours at a time, with only intermittent usage. Users expect to return to desktop applications after hours of no use and start working again.

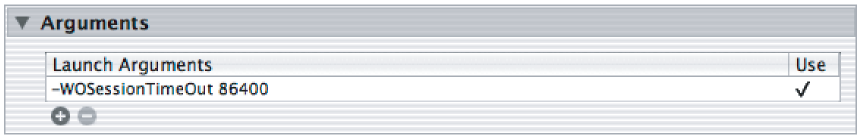
The default session timeout (60 minutes) is too short, so you need to set the timeout higher. Setting the timeout to 24 hours (86400 seconds) will better match the usage pattern of Java Client applications.

Follow these steps to change the session timeout:

1. Choose Edit Active Executable from the Project menu.

2. In the Arguments section, click the plus-sign button and enter `-WOSessionTimeOut 86400` as a launch argument as shown in Figure 3-17 (page 67).

Figure 3-17 Add session timeout launch argument



Launch Arguments	Use
-WOSessionTimeOut 86400	✓

More About Session Timeouts

What happens when the session times out and a client application is still running? The next time the client tries to connect to the server (either to save or retrieve data or when a request is made to the rule system), an error dialog appears noting that the session timed out and that any data not saved before the timeout was lost.

The dialog is modal, so the user has no choice but to quit the client application, and there is no way to reconnect except by restarting the client application.

You could implement an auto-save feature whereby the client application would display a warning dialog shortly before the session times out. Or, the client could just automatically save changes shortly before timeout. You would have to write code to poll for the timeout and implement the method `EOEditingContext.saveChanges()` accordingly.

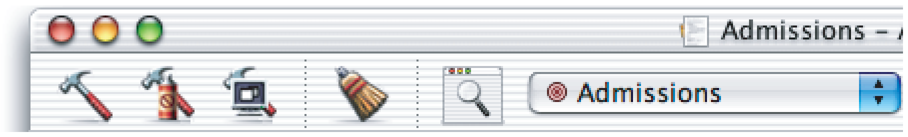
Note: When you deploy a Java Client application, you must set the session timeout in Monitor. Launch arguments set in Project Builder apply only to projects in development mode.

Build the Executable

You build a Java Client application using Project Builder. It handles everything for you, including specifying the correct Java classpath, configuring makefiles, creating directories, setting permissions, and so on.

1. Make sure that the Admissions target is selected in the Target pop-up menu as shown in Figure 3-18.

Figure 3-18 Select the Admissions target



2. Click the Build button (the one with the hammer) in the toolbar to build the application. The Build pane slides down and displays all console messages during the build, including any errors.

Run the Client Application

A Java Client application is made up of two parts: a server-side application and a client-side applet or application. You start the server application as you do any WebObjects application using one of these techniques:

- using Project Builder (during development)
- from the command line

- using Monitor (the preferred deployment mechanism)

The book *WebObjects Deployment Guide Using JavaMonitor* covers the second and third options. You can run the server application from Project Builder by clicking the Launch icon or choosing Run Executable from the Debug menu.

By default, Project Builder in Mac OS X runs the client application as a Java desktop application. However, there are many other ways to run the client application. By entering the application URL in a Web browser, you can start the client application as a Web Start application. You can also start applications from the command line or use the client launch script as described later in this section.

Prepare to Run the Project

In Mac OS X with WebObjects 5.1 and later, the client application is automatically started once the server application is up and running. So if you are developing with that configuration, you can skip this section and continue with "Using the Application" (page 71).

The `-WOAutoOpenClientApplication` flag (which, if not present in the launch arguments assumes the `YES` flag on development systems only) tells Project Builder to run the client launch script, which opens the client application as a Java desktop application.

The other methods of running the client application require some tweaks to the project. Add these launch arguments to make running the project manually a bit easier (add them to the same line as the `WOSessionTimeOut` argument):

```
-WOAutoOpenClientApplication NO -WOAutoOpenInBrowser NO -WOPort 8888
```

`-WOAutoOpenClientApplication NO` tells Project Builder to not automatically start the client application as a Java desktop application. Add this flag only if you always want to start the client application manually. When this feature is disabled, Project Builder automatically starts the client application as an applet in a Web browser unless it finds the `-WOAutoOpenInBrowser NO` launch argument. Although you can deploy Java Client application as applets, it's easier and often faster to deploy them as desktop applications during the development process.

By default, WebObjects runs applications on different ports each time they are run. During development, it's more convenient to use the same port; you can set a fixed port number using `WOPort`. Any arbitrarily high number (8888) is valid, but avoid common ports like 23 (telnet) and 80 (HTTP).

Client Launch Script

The client launch script is available only in Mac OS X. On WebObjects 5.1 running in Mac OS X, the `-WOAutoOpenClientApplication` flag invokes the client launch script automatically.

In Mac OS X, Project Builder creates a client launch script that includes all the classpath and executable information. All you need to feed it is the application URL. The launch script is named after your project, with a `_Client` suffix. It's located in your application's `.woa` in the `Contents/MacOS` directory. By default, an application's `.woa` file is in the `build` directory in the project's root directory.

To run the application:

1. Open a Terminal shell and `cd` to that directory (`Admissions.woa/Contents/MacOS`).

2. Copy the application URL from the Run pane in Project Builder.

3. At the shell prompt, paste the URL after entering the following:

   ```
   ./Admissions_Client
   ```

   The complete shell command to run the script is `./Admissions_Client http://localhost:8888/cgi-bin/WebObjects/Admissions`. Alternatively, you can enter the command with the full pathname from any directory in the shell: `~/Projects/Admissions/build/Admissions.woa/Contents/MacOS/Admissions_Client http://localhost:8888/cgi-bin/WebObjects/Admissions`

The Java virtual machine starts up, and in a few moments, the Direct to Java Client application is ready to use.

Behind the Steps

Step 3. The "./" command followed by a script name tells the shell to look for the script name starting in the current directory. The client launch script is simply a shell script, and you may want to open it in a text editor to see exactly what it does.

Java

To start the client as a stand-alone Java application outside a browser, use the `java` command-line tool. The syntax for starting a Java Client application is

```
java -classpath path
com.webobjects.eoapplication.EOApplication
-applicationURL url
-page pageName
```

The `classpath` argument must specify all the Enterprise Object classes and your custom classes. Fortunately, the `wojavaclient.jar` file includes all the Enterprise Object classes the client needs, so you simply need to specify its location in the `classpath` argument.

The `applicationURL` argument specifies the URL to connect to, which is displayed in the server application's console after initialization. The `page` argument specifies the name of the page that contains the WOJavaClientComponent. If it is not specified, "Main" is assumed, which is the default. Here's an example:

```
[trivium] brent% java -classpath /System/Library/Java/wojavaclient.jar
com.webobjects.eoapplication.EOApplication -applicationURL http://
trivium.apple.com:8888/cgi-bin/WebObjects/Admissions
```

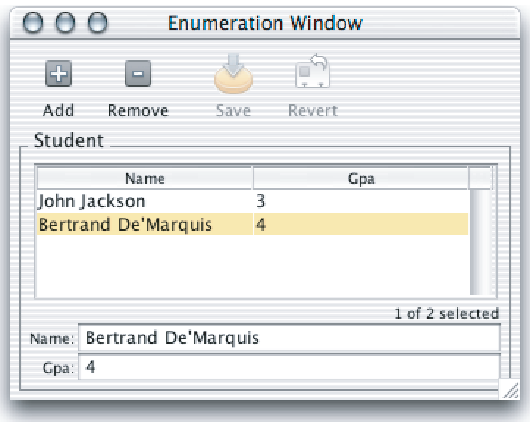
Mac OS X Application

In Mac OS X, you can package Java Cilent applications as double-clickable desktop applications. This deployment method is not practical during development and is more appropriate during deployment. See "Desktop Applications" (page 173) for instructions.

Using the Application

When you launch a Direct to Java Client application, the generation layer analyzes your EOModel and generates the user interface accordingly. Currently, your model and database have only a single table, and by default, Direct to Java Client displays a window to enumerate that table, as shown in Figure 3-19 (page 71).

Figure 3-19 Default enumeration window



Name	Gpa
John Jackson	3
Bertrand De'Marquis	4

1 of 2 selected

Name: Bertrand De'Marquis
Gpa: 4

You can add, delete, and save records, as well as revert changes made since the last save. You can also rearrange the columns.

So far, you haven't written a single line of code, yet the Enterprise Object technology has provided the following for you:

- automatic primary-key generation when you insert new objects
- communication between client and server
- coordination between user interface and data store

Notice that only the attributes you marked as client-side class properties are displayed in the client. `studentID`, the entity's primary key, isn't displayed since it wasn't marked as a client-side class property in the EOModel.

There is a significant problem with the GPA field. You'll notice that decimal points are automatically truncated, which is unacceptable when recording GPAs. This is due to that field's data type: `int` (external) and `Integer` (internal).

Since uncustomized Direct to Java Client user interfaces are contingent on the contents of their corresponding EOModels, you need to edit the EOModel to correct this problem.

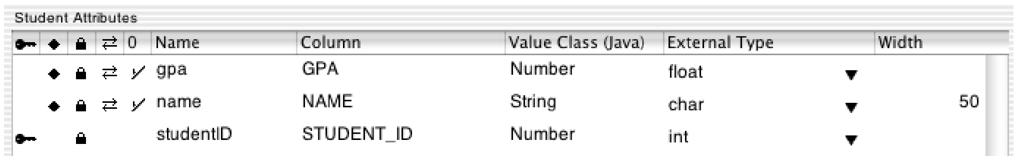
Follow these steps to edit the EOModel:

1. First, quit the client application by choosing Quit from the File menu, and quit the server application by clicking the Stop button in the Project Builder toolbar.

2. When you created the project, Project Builder made a copy of the EOModel and put it in the project directory. So, you must edit that copy. Find `Admissions.eomodeld` in the Resources group and double-click it to open it in EOModeler.

3. Change the external type for the `gpa` attribute to `float`. You can do this with the Inspector or in the table view. In the Inspector, change the internal data type to `Double`. The model should now resemble Figure 3-20.

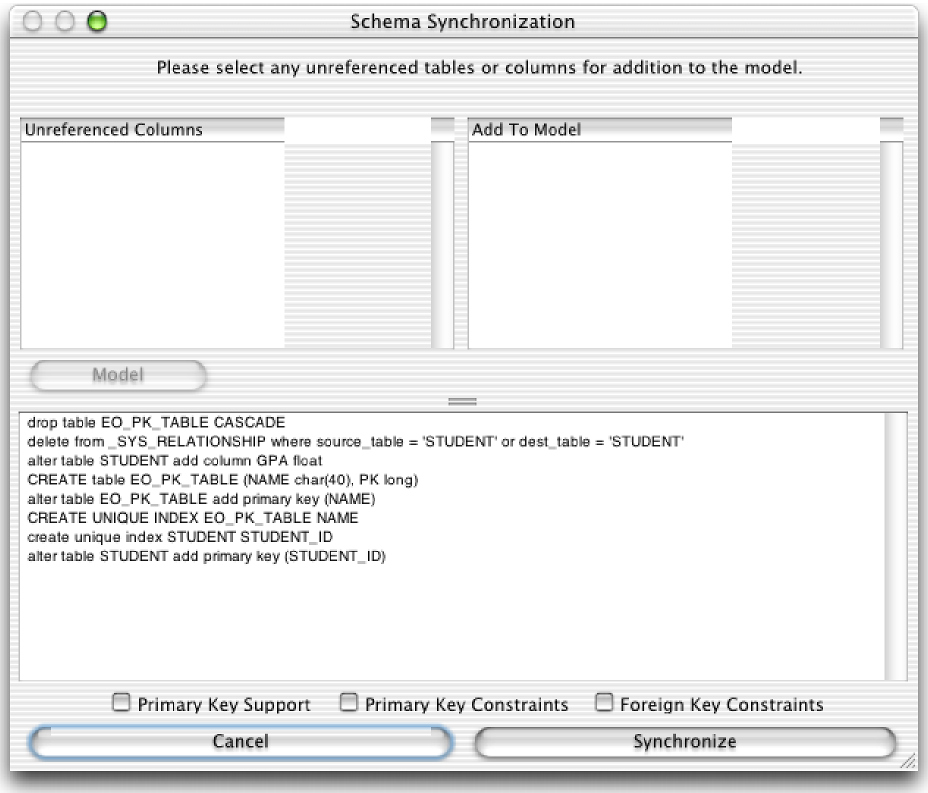
Figure 3-20 Revised model


Student Attributes

				0	Name	Column	Value Class (Java)	External Type	Width
	◆	🔒	⇄	✓	gpa	GPA	Number	float ▼	
	◆	🔒	⇄	✓	name	NAME	String	char ▼	50
🔑		🔒			studentID	STUDENT_ID	Number	int ▼	

4. External types are database-specific, so you need to synchronize the model with the database. Save the model, then select the root of the entity tree (Admissions) and choose Synchronize Schema from the Model menu. Deselect all three options in the Schema Synchronization window as shown in Figure 3-21. Then click Synchronize.

Figure 3-21 Schema Synchronization window


Schema Synchronization
Please select any unreferenced tables or columns for addition to the model.
Unreferenced Columns
Add To Model
Model

```
drop table EO_PK_TABLE CASCADE
delete from _SYS_RELATIONSHIP where source_table = 'STUDENT' or dest_table = 'STUDENT'
alter table STUDENT add column GPA float
CREATE table EO_PK_TABLE (NAME char(40), PK long)
alter table EO_PK_TABLE add primary key (NAME)
CREATE UNIQUE INDEX EO_PK_TABLE NAME
create unique index STUDENT STUDENT_ID
alter table STUDENT add primary key (STUDENT_ID)
```

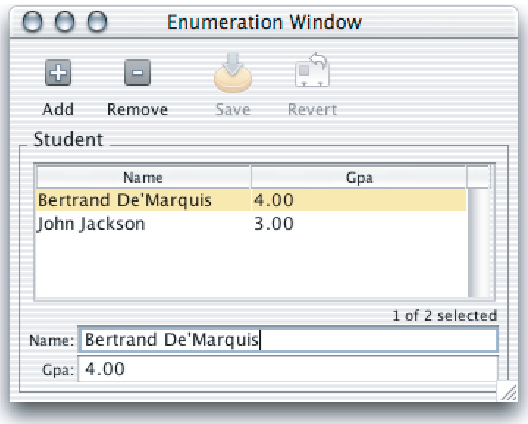
☐ Primary Key Support ☐ Primary Key Constraints ☐ Foreign Key Constraints

5. In OpenBase Manager, verify that the data type for the `gpa` attribute changed. Do this by choosing the Admissions database and clicking the Schema Design button to view the database's tables.

6. Save the model, build the project, and run both the client and server applications.

7. Enter a few new records, and save changes. Notice how decimals are now preserved as shown in Figure 3-22.

Figure 3-22 Revised enumeration window



Name	Gpa
Bertrand De'Marquis	4.00
John Jackson	3.00

Name: Bertrand De'Marquis
Gpa: 4.00

Behind the Steps

Step 4: Many model modifications do not require you to synchronize the schema. However, anytime you add, remove, or change the name or type of an attribute, synchronization is necessary.

Customizing the Application

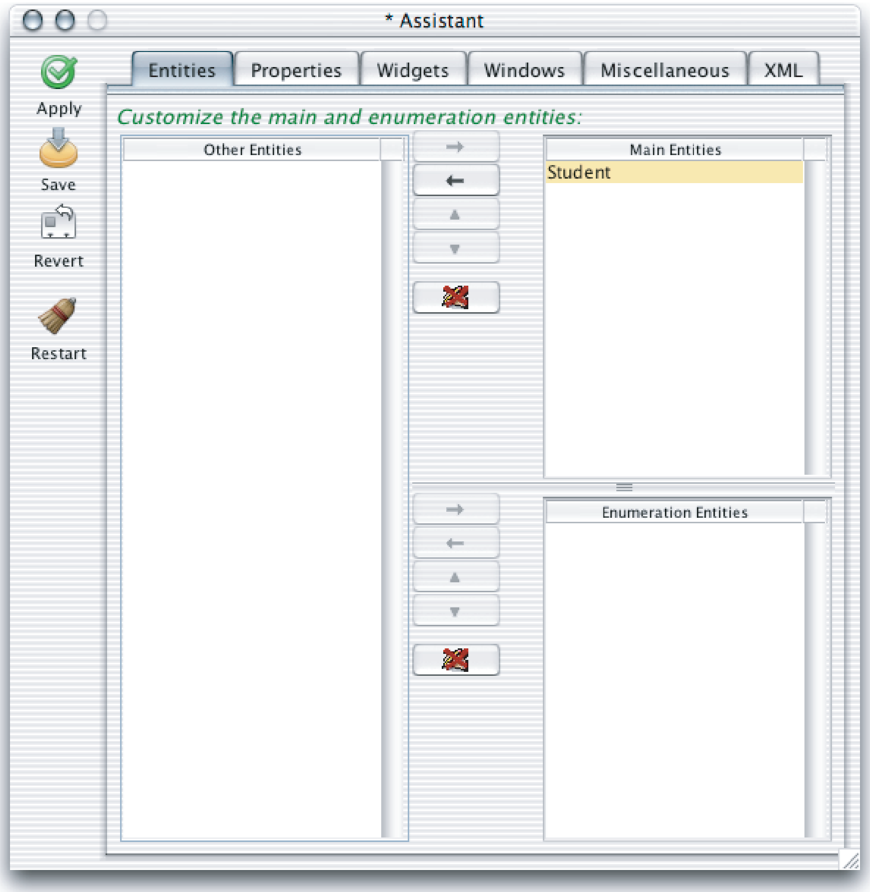
There are many ways to customize Direct to Java Client applications, including the tool Direct to Java Client Assistant. Assistant is a Java application included in every Direct to Java Client application, and it provides an easy way to perform simple customizations. It is described in depth in "Inside Assistant" (page 85).

The following steps introduce you to Assistant:

1. While the Direct to Java Client application is running, select Assistant from the Tools menu.

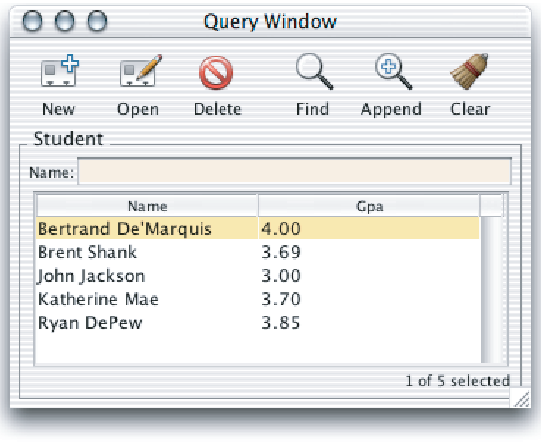
2. Change the Student entity from an Enumeration entity to a Main entity as shown in Figure 3-23.

Figure 3-23 Change entity type



3. Click Save, then Restart to see how the window type changes. You can now search the database using a query string. Alternatively, you can fetch all records in the database by clicking the Find button without entering a query string, as shown in Figure 3-24.

Figure 3-24 Query window with data

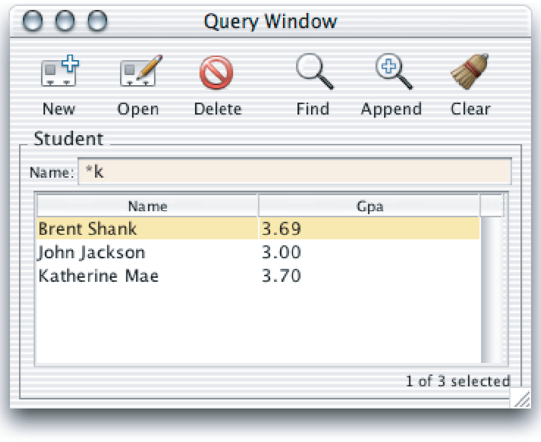


Name	Gpa
Bertrand De'Marquis	4.00
Brent Shank	3.69
John Jackson	3.00
Katherine Mae	3.70
Ryan DePew	3.85

1 of 5 selected

4. Click New to add records; then enter different query strings to test the application. Figure 3-25 (page 75) illustrates a query for names containing "k."

Figure 3-25 Query window searching for names containing "k"



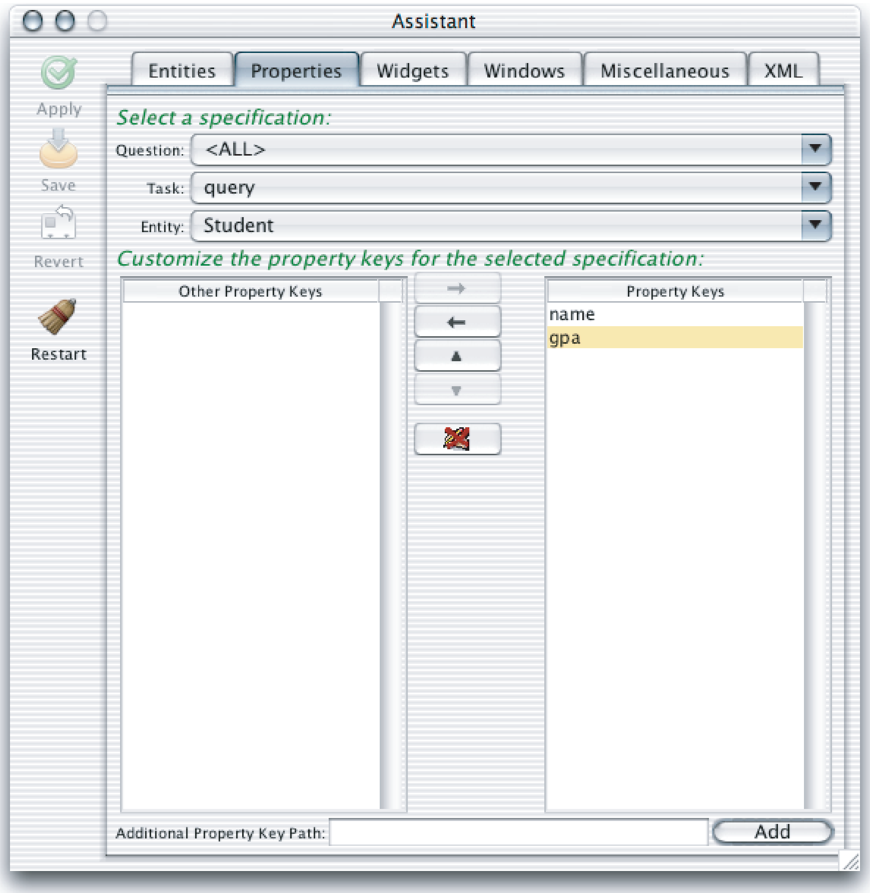
Name: *k

Name	Gpa
Brent Shank	3.69
John Jackson	3.00
Katherine Mae	3.70

1 of 3 selected

5. You might want to also query on the `gpa` attribute. In Assistant, switch to the Properties pane, and choose "query" in the Task pop-up menu. You'll notice that the `gpa` property key is listed in the Other Property Keys list. Move it to the Property Keys list as shown in Figure 3-26 (page 76), save, and restart the application. You can now query on the `gpa` field also, as illustrated in Figure 3-27 (page 76). By default, the application provides two fields so you can search for a range of GPAs.

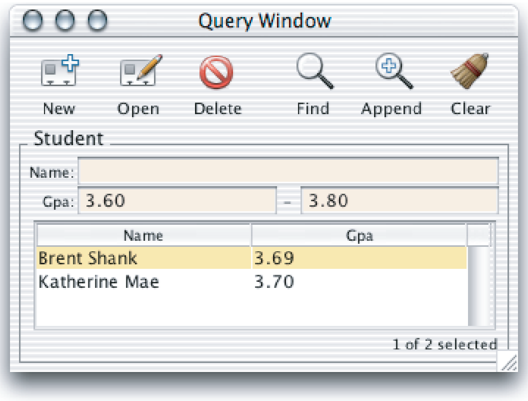
Figure 3-26 Properties pane in Assistant



Question: <ALL>
Task: query
Entity: Student

Property Keys
name
gpa

Figure 3-27 Query on GPA



Name:
Gpa: 3.60 - 3.80

Name	Gpa
Brent Shank	3.69
Katherine Mae	3.70

6. You should probably improve the label for the `gpa` field. It should be all capitals. In Assistant, switch to the Widgets pane. Make sure the Property Key pop-up menu reads “gpa.” Under Customize Widget Parameters, change the Label field to `GPA`. Save and restart the application. Notice how the widget label changed.

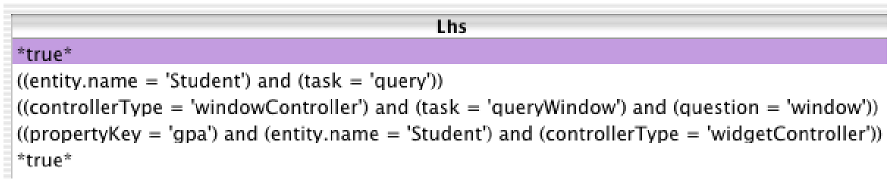
7. In Assistant, switch to the Windows tab and change the window label to `Admissions`. Save and restart to see the changes.

8. Direct to Java Client user interfaces are defined in XML descriptions. The XML pane in Assistant displays the XML descriptions for the various specifications in an application. Switch to the XML pane and browse the specifications in the current application.

The changes you made in Assistant are stored in the project’s `user.d2wmodel` file. Open this file in Rule Editor to see the rules that were created when you made changes using Assistant. The Rule Editor application is installed in `/Developer/Applications`.

Figure 3-28 shows the left-hand side or conditional of each rule that Assistant created as you customized the application. It says “if the application is in this state, fire the rule and resolve the rule’s right-hand side.”

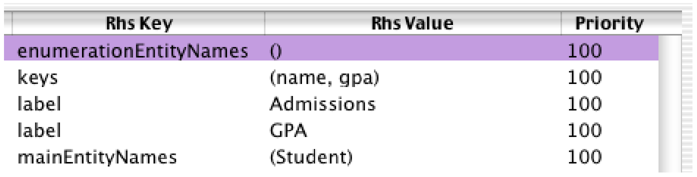
Figure 3-28 Left-hand side of rules



Lhs
true
((entity.name = 'Student') and (task = 'query'))
((controllerType = 'windowController') and (task = 'queryWindow') and (question = 'window'))
((propertyKey = 'gpa') and (entity.name = 'Student') and (controllerType = 'widgetController'))
true

Figure 3-29 shows the right-hand side of each rule. The first rule says that none of the entities in the application are considered enumeration entities. The second rule says to provide fields in query windows for both the `name` and `gpa` properties of the Student entity. The third rule says to use the label “Admissions” for query windows for the Student entity. The fourth rule says to use the label “GPA” for the property key label for the `gpa` attribute of the entity `Student`. The fifth rules says that the Student entity is a main entity.

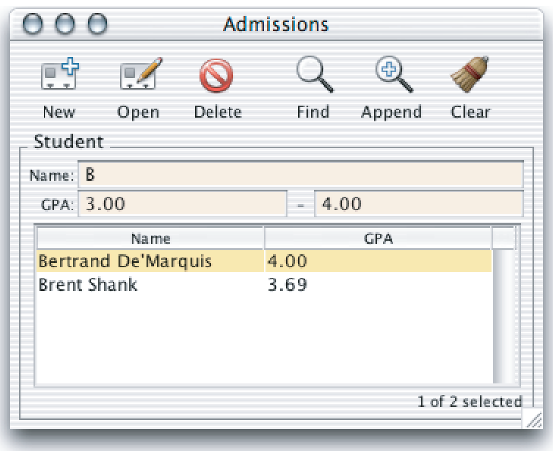
Figure 3-29 Right-hand side of rules



Rhs Key	Rhs Value	Priority
enumerationEntityNames	()	100
keys	(name, gpa)	100
label	Admissions	100
label	GPA	100
mainEntityNames	(Student)	100

At this point, your application should resemble Figure 3-30.

Figure 3-30 The application with simple customizations



Name	GPA
Bertrand De'Marquis	4.00
Brent Shank	3.69

1 of 2 selected

Behind the Steps

Step 1: To disable Assistant in the client application, pass `-EOAssistantEnabled NO` as a launch argument for the server application.

Assistant is available only when rapid turnaround mode is enabled (it is enabled by default on development systems). Rapid turnaround mode allows the application to access resources from the project directory rather than from the `.woa` bundle, which eases development and testing. Also, in Mac OS X, Assistant runs only if the project is open in Project Builder. Assistant needs access to write out the `user.d2wmodel` file, and it can do this only while the project is open.

Step 2: The Entities pane is selected by default. The entity type determines the window type. That is, each entity type has a default window type. Enumeration entities are represented by Enumeration windows; Main entities are represented by Query windows; Other entities are not represented by any particular window type. The rule system, which will be discussed later on at length, determines these rules.

Add a Relationship

Now that you're familiar with Direct to Java Client, you need to expand your EOModel so you can use more of its features. You'll add a new relationship representing a student's extracurricular activities.

Add an Entity

To create a new relationship, you need more than one entity. Quit the client application, stop the server application, and open the `Admissions.eomodeld` file from within Project Builder. In EOModeler, complete the following steps to enhance the model:

1. Add a new entity named "Activity" with table name "ACTIVITY". Its class is EOGenericRecord.

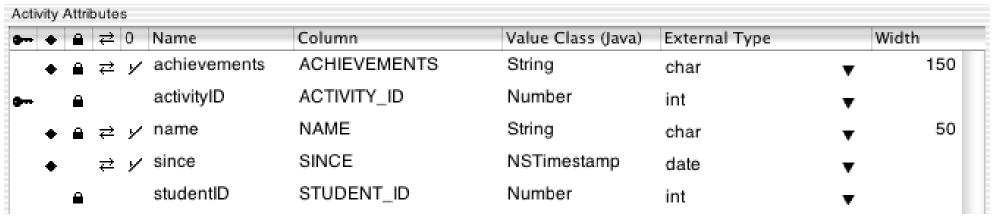
2. Add new attributes:

- Name: `activityID`; Column: ACTIVITY_ID; External Type: `int`; Internal Data Type: `Integer`. Do not make this a client-side class property or a server-side class property.
- Name: `name`; Column: NAME; External Type: `char`; Internal Data Type: `String`, width 50. Make this a client-side class property. Verify that this attribute is also marked as a server-side class property.
- Name: `achievements`; Column: ACHIEVEMENTS; External Type: `char`; Internal Data Type: `String`, width 150. Make this a client-side class property. Verify that this attribute is also marked as a server-side class property.
- Name: `since`; Column: SINCE; External Type: `date`; Internal Data Type: `Date`; Make this a client-side class property. Verify that this attribute is also marked as a server-side class property. *Don't lock on this attribute: Deselect the lock icon to the left of the attribute to do this.*

3. Make `activityID` the primary key in the Activity entity by clicking in the key column.

4. Add a foreign key by copying Student's primary key (`studentID`) into the Activity table. Do this by selecting `studentID` in the Student table, then choose Copy from the Edit menu, then click in the Activity table, and choose Paste from the Edit menu. Verify that `Activity.studentID` is not marked as a primary key, as a server-side class property, or as a client-side class property in the Activity entity. The new entity should look as shown in Figure 3-31 (page 79).

Figure 3-31 Activity entity


Activity Attributes

🔑	◆	🔒	⇄	0	Name	Column	Value Class (Java)	External Type	Width
	◆	🔒	⇄	✓	achievements	ACHIEVEMENTS	String	char	150
🔑		🔒			activityID	ACTIVITY_ID	Number	int	
	◆	🔒	⇄	✓	name	NAME	String	char	50
	◆		⇄	✓	since	SINCE	NSTimestamp	date	
		🔒			studentID	STUDENT_ID	Number	int	

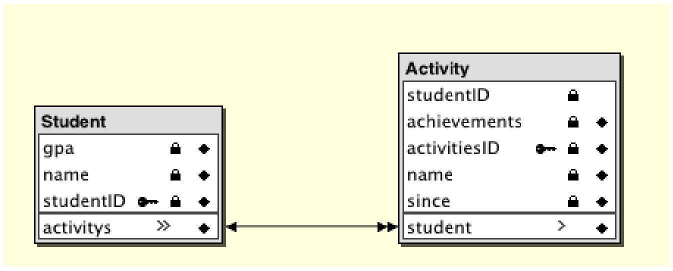
5. Select the Activity entity in the tree view and choose Generate SQL from the Property menu. Since you already generated primary key support the first time you generated SQL, make sure to deselect the option Create Primary Key Support. Only Create Tables and Primary Key Constraints should be selected.

Make the Relationship

The relationship you'll add to the model is a **one-to-many** relationship. That is, one Student object can be related to many Activity objects. In most cases, to-many relationships need at least a foreign key and a primary key. These keys are the attributes on which the relationship joins. Follow these steps to form a relationship between Student and Activity:

1. In diagram view (Tools > Diagram View), Control-drag from Student's primary key (`studentID`) to Activity's foreign key (`studentID`) as shown in Figure 3-32 (page 80).

   Figure 3-32 Relate Student and Activity

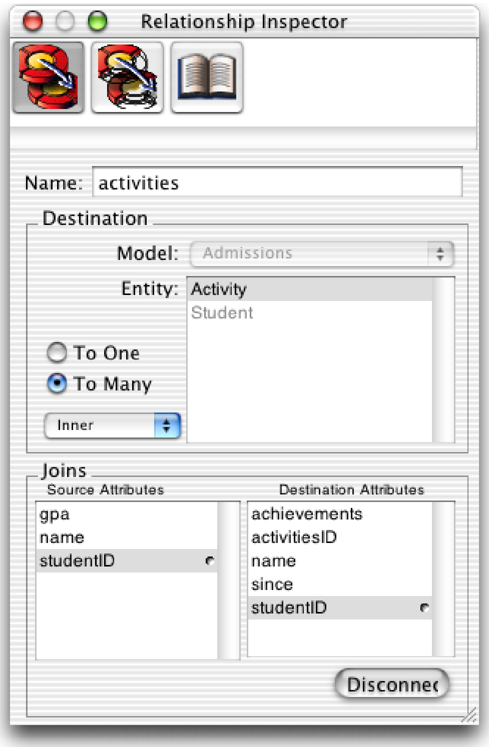
   

   This action creates a relationship in both entities: a to-many relationship from Student to Activity and a to-one relationship from Activity to Student.

2. The Relationship Inspector allows you to customize the relationship. If it is not visible on the screen, select Inspector from the Tools menu.

   Change the relationship name to "activities."

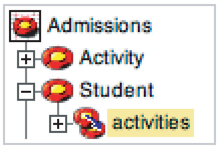
   Figure 3-33 Relationship Inspector for Student's `activities` relationship

   Name: activities
   Destination
   Model: Admissions
   Entity: Activity
   Student
   To One [✓ To Many]
   Inner
   Joins

Source Attributes	Destination Attributes
gpa	achievements
name	activitiesID
studentID	name
	since
	studentID

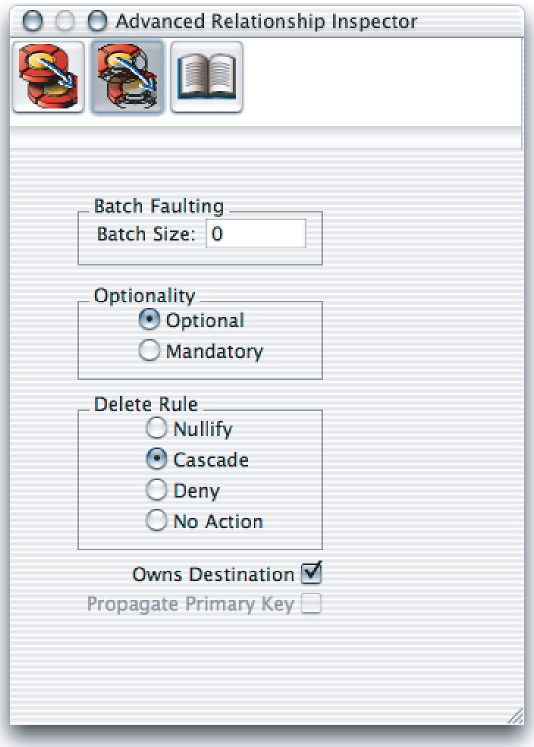
3. As each student can have multiple activities, the relationship from the Student entity to the Activity entity is a to-many relationship. Click the plus sign next to the Student entity in the tree view to show its relationship, `activities`.



Select the relationship and in the inspector, make sure To Many is selected and that `studentID` is selected in both the Source Attributes list and the Destination Attributes list, as shown in Figure 3-33 (page 80).

4. Each activity is specific to a particular student so you need to establish ownership and a delete rule that reflects this business logic. With the `activities` relationship selected, switch to the Advanced Relationship Inspector. Select Cascade as the delete rule and select the box labeled Owns Destination, as shown in Figure 3-34.

Figure 3-34 Configure ownership and delete rule


Advanced Relationship Inspector

Batch Faulting
Batch Size: 0

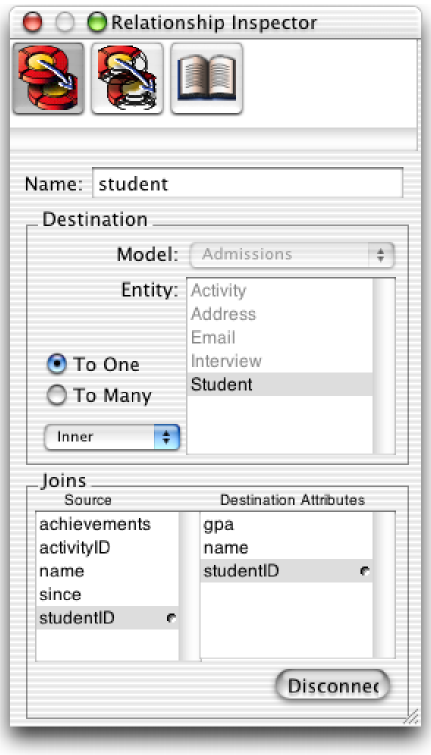
Optionality
[✓ Optional] Mandatory

Delete Rule
Nullify [✓ Cascade] Deny No Action

Owns Destination ✓
Propagate Primary Key ☐

5. Each activity and its attributes are unique to a single student, so the relationship from Activity to Student is a to-one relationship. Click the plus sign next to the Activity entity in the tree view to show its relationship, `student`. Select the relationship and in the inspector, make sure To One is selected. Also verify that `studentID` is selected in both attributes lists as shown in Figure 3-35 (page 82).

Figure 3-35 Relationship Inspector for Activity's `student` relationship



Name: student

Model: Admissions

Entity: Activity, Address, Email, Interview, Student (selected)

[✓ To One] To Many

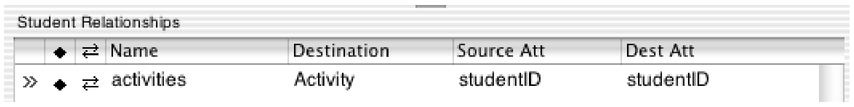
Inner

Source	Destination Attributes
achievements	gpa
activityID	name
name	studentID (selected)
since	
studentID (selected)	

6. As with entity attributes, you can choose to make relationships available to the client application. You need to add the new relationships as client-side class properties. Switch to table mode (Tools > Table Mode). Below the attributes table is a table for relationships. You may need to add the client-side class properties column to the relationship view.

   If the Student entity is selected in the tree view, its relationship (`activities`) is shown in this table, as Figure 3-36 (page 83) illustrates. Selecting the Activity entity displays its relationship (`student`), as Figure 3-37 (page 83) illustrates. Make the `activities` relationship in the Student entity a client-side class property by clicking in the double-arrow column to the left of it, as shown in Figure 3-36 (page 83). However, do not make the `student` relationship in the Activity entity a client-side class property, as shown in Figure 3-37 (page 83).

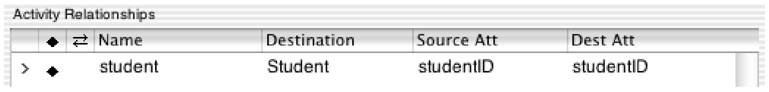
Figure 3-36 Make Student to Activity relationship a client-side class property



Student Relationships

	◆	⇄	Name	Destination	Source Att	Dest Att
»	◆	⇄	activities	Activity	studentID	studentID

Figure 3-37 Do not make Activity to Student relationship a client-side class property



Activity Relationships

	◆	⇄	Name	Destination	Source Att	Dest Att
›	◆		student	Student	studentID	studentID

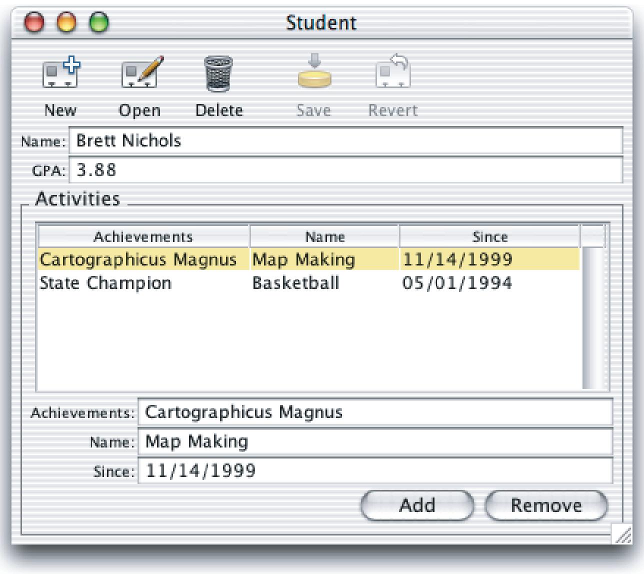
7. Save the model.

You do not need to synchronize the schema as the Enterprise Object technology manages the relationships for you. This helps you build reusable enterprise object models since the relationship is not database-specific.

The Enhanced Application

Build the project and run both the client and server applications. Direct to Java Client analyzes the altered EOModel file and generates the user interface based on the new relationship. Now, when you make a new Student record you can also add activities for that student as shown in Figure 3-38 (page 83).

Figure 3-38 Add activities to new Student record



Name: Brett Nichols
GPA: 3.88

Activities

Achievements	Name	Since
Cartographicus Magnus	Map Making	11/14/1999
State Champion	Basketball	05/01/1994

Achievements: Cartographicus Magnus
Name: Map Making
Since: 11/14/1999

When you added the Activity entity to the model and related it to the Student entity, you changed how the rule system considers the two entities. They are both now considered Main entities. However, this doesn't precisely fit the application's requirements. The Student entity's `activities` relationship is a master-detail relationship in which Student objects are the master objects and Activity objects are the detail objects. Each Activity record is specific to a single student so that you can only add Activity records from within a Student record—you cannot add Activity records that aren't immediately related to a Student record.

For the rule system to provide a user interface that reflects this relationship, you need to explicitly make the Activity entity an "other" entity in Assistant. Doing this allows you to add activities to a Student record by clicking the Add button in form windows for the Student entity, but it doesn't allow users to query on Activity records or add Activity records outside the scope of a Student record.

If you make Activity an enumeration or a main entity using Assistant, the application provides different mechanisms to add activities to student records. Experiment with this by changing Activity's entity type in Assistant and restarting the client application.

Make sure to change the Activity entity back to an "other" entity to successfully complete the other tutorials.

Where to Go From Here

"The Distribution Layer" (page 93) provides an important overview of how client-server communication in Java Client applications works. Some of the concepts in that chapter are put into practice in "Enhancing the Application" (page 103) However, you may want to continue with the second tutorial and then read the chapter on the distribution layer as it assumes a deeper understanding of Java Client concepts that you'll learn in "Enhancing the Application" (page 103).

Inside Assistant

This chapter gives more details about the Direct to Java Client Assistant, the first tool to use for customizing Direct to Java Client applications. Assistant is an application that helps you write rules for Direct to Java Client applications. It edits the `user.d2wmodel` file that all Direct to Java Client applications have. Assistant does nothing special that you cannot do by hand in Rule Editor but it is saves you from needing to know all the keys , values, and valid key-value combinations in the rule system.

This chapter explains what you can do in each of the panes in the Direct to Java Client Assistant. This chapter does not explain the basics of the rule system. For that information, see "Inside the Rule System" (page 163).

Entities Pane

Direct to Java Client defines three entity types: main, enumeration, and other. An entity can be only one of these types. The Entities pane provides an easy way for you to change entities from one type to another.

Although you can freely change an entity's type, you should be cautious when doing so. The rule system considers an entity to be of a particular type based on certain heuristics (these are defined for each entity in the following sections). This means that when you change an entity's type, you are changing what the rule system expects an entity to be. That is, you are changing how the rule system expects an entity to be defined in an EOModel.

If you need to change an entity's type, you should first consider changing the application's data model so that the rule system considers a particular entity to be of the type you want by default. The rule system is smart and if it interprets your data model in ways you don't expect, it's highly likely that your data model is misconfigured.

For example, if you want the rule system to consider a particular entity an enumeration entity, make sure that the entity has fewer than five attributes and that it is not the destination of any relationships that use the cascade delete rule. The rule system then by default considers the entity to be an enumeration entity. (Again, how the rule system determines an entity's type is defined in the following sections).

In summary, although Assistant allows you to easily change an entity's type, an entity's type should reflect the entity's characteristics as defined in the EOModel in which the entity is defined.

Main Entities

A main entity is generally a top-level entity that users work with most frequently. Consequently, Direct to Java Client creates a tab view for each main entity in the Query Window and provides form windows for editing each of the main entities.

Direct to Java Client by default defines a main entity as one that is not the destination of any relationships that have the following characteristics:

- propagate primary key
- own destination
- use the cascade delete rule

If you're using entity inheritance, abstract entities are not considered main entities by default.

Enumeration Entities

By default, an enumeration entity conforms to the conditions for main entities and additionally conforms to these conditions:

- the entity has fewer than five attributes
- the entity has no relationships that are mandatory
- all the entity's relationships use the deny delete rule

In practice, enumeration entities should define a collection of values that represent a list of choices. The values in enumeration entities are usually fairly static, and you usually don't want a complex user interface for changing them. Enumeration entities exist to provide a simple user interface for a list of choices.

An enumeration window contains a tab view for each of the application's enumeration entities. The tab view for a particular entity shows the complete set of values in that entity. You can add a new value to the enumeration's collection (Add), delete a value from the collection (Remove), and modify a value (make changes and click Save).

"Other" Entities

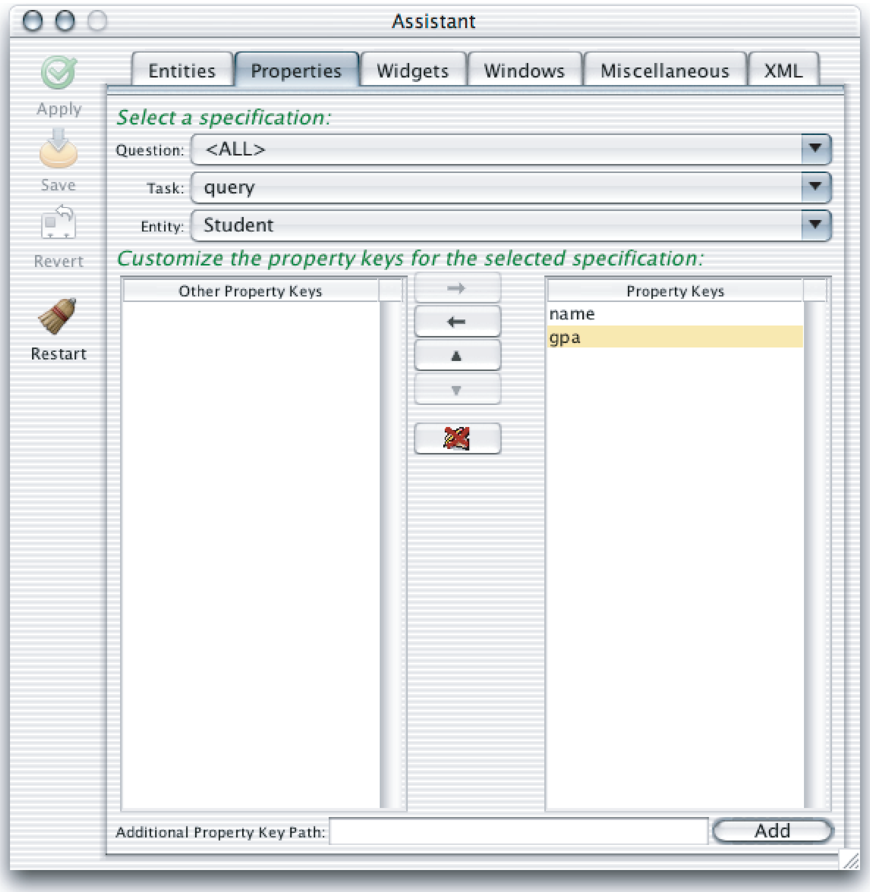
Entities that aren't main or enumeration entities are considered other entities. Other entities can be manipulated through the master-detail user interfaces of main entities. You cannot directly query or enumerate on other entities: A tab view is not provided for them in query windows or in enumeration windows. The only way you can access an other entity is through a master-detail interface in a main entity in which the other entity is the detail interface. A master-detail interface represents a relationship in which the master object (such as a Student) has one or many detail objects (such as Activities).

Properties Pane

The Properties pane lets you see how the rule system interprets the attributes of entities in the application's data model. It simply tells you which attributes of each entity are displayed for each window type and each task in the client application. The attributes of a particular entity that are displayed for a particular window type and task are listed in the Property Keys list.

Figure 4-1 shows that for the Student entity for the query task for all question types (window or modal dialog, which simply represents the type of the window that is displayed) to display the `name` and `gpa` attributes. Figure 3-30 (page 78) shows a window that reflects this configuration. The window is a nonmodal window that is a query window that queries on the Student entity and provides fields for qualifying the query on the `name` and `gpa` attributes.

Figure 4-1 Properties pane in Assistant



Select a specification:
Question: <ALL>
Task: query
Entity: Student
Customize the property keys for the selected specification:
Property Keys: name, gpa
Additional Property Key Path:

Task Pop-Up Menu

The Task pop-up menu lists the four primary tasks in the client application: form, identify, list, and query. When you choose one of these tasks from the menu, the properties that are displayed for each entity for that task are shown in the Property Keys list. By selecting an entity, a task, and a question, you can choose which properties (attributes) are displayed in form windows (form task), in modal dialogs (identify task), in query windows (query task), and data lists (list task).

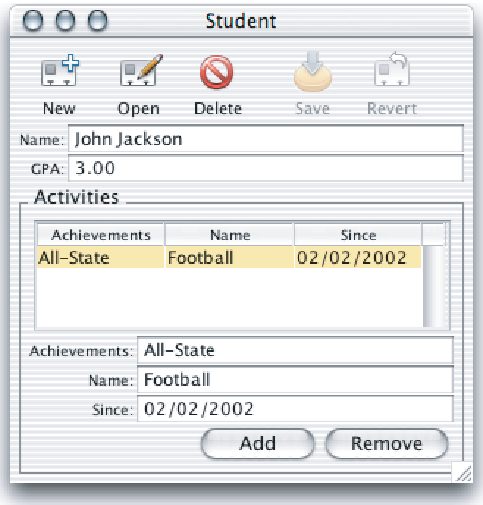
The different tasks correspond to four different window types:

form

Used to enter new records or edit existing records. Contains a property key for each attribute of an entity that is a property key. Each property key is associated with a widget such as a text field or checkbox.

Figure 4-2 shows a form window for an entity named Student, which has two attributes that are in the Property Keys list in Assistant.

Figure 4-2 Form task



Name:	John Jackson
GPA:	3.00

Activities

Achievements	Name	Since
All-State	Football	02/02/2002

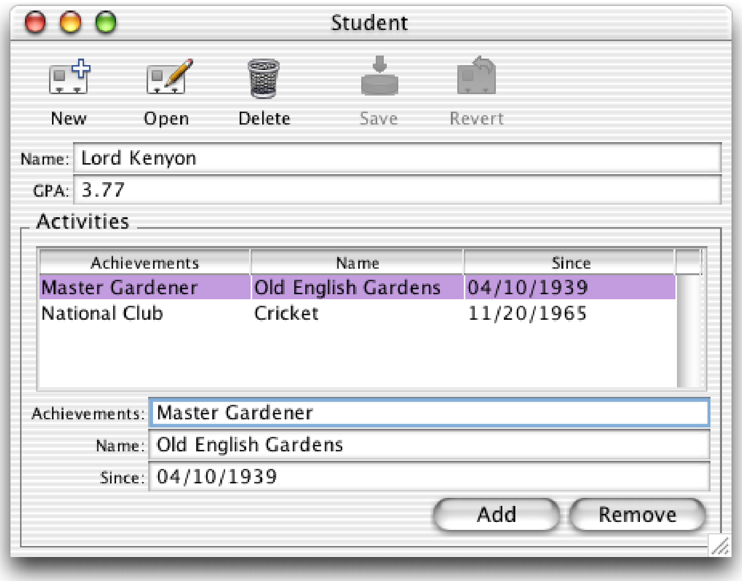
Achievements:	All-State
Name:	Football
Since:	02/02/2002

identify

Used whenever an object is referenced in the user interface, such as in a master-detail interface. Figure 4-3 shows a form window that displays a master-detail relationship. You'll add this relationship later in the chapter.

The student record is the master record, while the student's activities are the detail records. The detail records are represented by the identify task in the rule system. So if you wanted to display only the `name` and `since` attributes of the activities records, you would select the identify task and the Activity entity in the Properties pane of Assistant and move all other attributes to the Other Property Keys list.

Figure 4-3 Identify task



Name:	Lord Kenyon
GPA:	3.77

Activities

Achievements	Name	Since
Master Gardener	Old English Gardens	04/10/1939
National Club	Cricket	11/20/1965

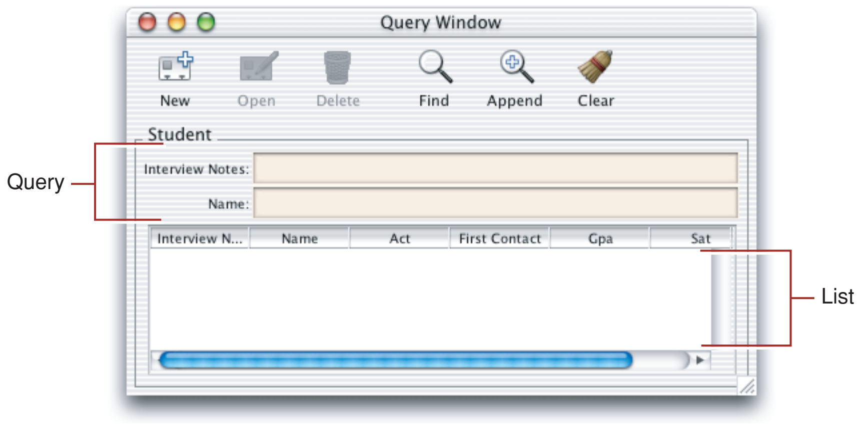
Achievements:	Master Gardener
Name:	Old English Gardens
Since:	04/10/1939

list
: Displays the results of a query in a table view. Only the attributes that are property keys are displayed in the list. Figure 4-4 shows a list task within a query window.

query
: Provides a text field to query on for each property key that can be queried on as a string. That is, class properties that map to binary data types cannot be directly queried on. For attributes that map to numeric data types, two text fields are provided so users can query on a range of numbers.

  Figure 4-4 shows a query window that contains a query task that maps to an entity named Student, which has two property keys, Interview Notes and Name.

Figure 4-4 Query window and list task



Question Pop-Up Menu

This menu allows you to change task behavior depending on the window type. By default, all changes to tasks affect both windows and modal dialogs. But if you want different behavior or a different look in one of the window types, choose it in the Question menu before making changes to the Task pop-up menu and Property Keys list.

Property Keys List

The property keys displayed in Assistant are all the attributes of an entity that are marked as client-side class properties in the application's EOModel. Only those attributes that are listed in the Property Keys list are visible in the client application. You can choose which attributes are displayed for a particular task for a particular entity by moving attributes to and from the Property Keys list.

Additional Property Key Path Field

A property key is by definition not just an attribute in an entity. A property key is any key in an enterprise object, which includes relationships, methods, and instance variables, that can be resolved by key-value coding.

By default, the property keys that are displayed in Assistant are the client-side properties (attributes and relationships) of the selected entity. It's common to write methods in business logic classes (custom enterprise object classes) that return a value.

To display this value in the client application, you need to let the rule system know about the method. You do this by adding an additional property key path using the Additional Property Key Path field. The rule system then looks in the custom enterprise object class with which the entity is associated for a key that corresponds to the new property key path. Again, this key can be any key that is resolvable through key-value coding.

Widgets Pane

The Widgets pane lets you tweak user interface elements by adjusting their editability, label, format, alignment, size, and more. You make adjustments on a per-task basis. By assigning new property keys to a particular widget type, you can easily add custom actions, QuickTime movies, and other user interface features to applications.

Starting with WebObjects 5.2, the Widgets pane allows you to specify layout hints and levels for property keys in form windows. You can specify the following layout hints (components are property key widgets):

- **Columns**: Components are placed underneath each other from top down and in multiple columns if there are many.
- **Row**: Components are placed side-by-side from left to right.
- **FullWidth**: Components are placed underneath each other and each covers the full width of the window.
- **Box**: Components are placed in a titled box.
- **Switch**: Components are placed in a switch view (typically a tab view).
- **Subwindow**: Components are placed in a subwindow and a button appears in the main window to display the subwindow.
- **Inspector**: Components are placed in an inspector window and a button appears in the main window to display the inspector.
- **VerticalSplit**: Components are placed in a vertical split view. You must change the layout hint for exactly two property keys to use VerticalSplit.
- **HorizontalSplit**: Components are placed in a horizontal split view. You must change the layout hint for exactly two property keys to use HorizontalSplit.

For particularly complex data models, you may also want to specify a layout level. Layout levels specify the order in which components are generated by the rule system. The rule system generates the user interface one level at a time, with the order of the layout hints as listed above. Layout levels allow you to order the visual display of properties without regard to the entity with which those properties are associated.

For example, a window would display at the top the properties for which you specified a layout level of 1 and a layout hint of Columns, below that the properties with the layout level 1 and the hint Row, and so on through the list of layout hints at that level, which ends with the hint HorizontalSplit.

After that are displayed the properties for which you specified a layout level of 2 and a layout hint of Columns, then the properties with the layout level 2 and hint Row, and so on through the list of layout hints.

Before layout levels were available, the only way to specify the visual layout of properties was in the Properties pane by ordering the Property Keys list. This provided no facility to control the order of entities in a particular layout or to intermix properties from one entity with another.

Windows Pane

You use the options in the Windows pane to change the window-level characteristics of specifications in the application. This allows you to specify whether a particular window is regular or modal, or is a dialog. You can also specify the title for each type of window, the default positioning, and minimum width and height.

The two options in the "Customize runtime behavior" section allow you to control how the controller factory caches the components it generates.

Factory Reuse Mode
: Tells the controller factory how and when to generate new windows. For example, the Enumeration Window has a factory reuse mode of AlwaysReuse whereas form windows have a factory reuse mode of NeverReuse (which tells the controller factory to open a new form window for each request for a form window).

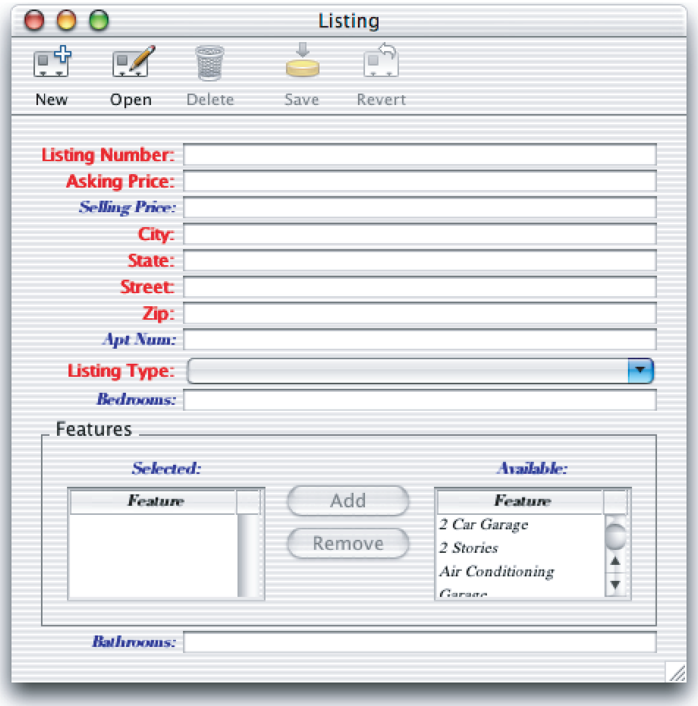
Dispose If Deactivated:
: Tells the window controller whether closing it should dispose of the window in memory (that is, if it should not make an attempt to prevent it from being garbage collected). For example, form windows just disappear whereas an Enumeration Window stays in memory as single instance.

Miscellaneous Pane

The Miscellaneous pane contains additional options for customizing the user interface. Starting with WebObjects 5.2, it offers considerably more options and allows you to take precise control over the user interface layout. Again, like everything throughout Assistant, the options in this pane generate rules rather than code.

Figure 4-5 shows the layout that results from many customizations in the Miscellaneous pane.

Figure 4-5 Very customized layout



Listing Number:
Asking Price:
Selling Price:
City:
State:
Street:
Zip:
Apt Num:
Listing Type:
Bedrooms:
Features
Selected:
Available:

Feature

Feature
2 Car Garage
2 Stories
Air Conditioning
Garage

Bathrooms:

The Optimize For Mac option generates a layout that adheres more closely to the Aqua Human Interface Guidelines. The Use Button and Use Action options control the placement and look of action buttons that activate subwindows, inspector windows, or other action controllers in the client application. For example, if you use the Inspector layout hint, you must also set Use Action to True and Use Button to False in order for that inspector's action button to appear in the toolbar.

XML Pane

The XML pane provides a list of all the specifications that make up the components in the client application. Specifications are XML hierarchies that are generated by the rule system and that are transformed into Swing by the generation layer on the client. You'll learn more about this in "Inside the Rule System" (page 163).

You use the information in this pane when freezing XML files as described in "Freezing XML User Interfaces" (page 199). The Save button saves the XML description as a text file.

The Distribution Layer

The distribution layer (`com.webobjects.eodistribution` and `com.webobjects.eodistribution.client`) consists of the objects that make client-server communication in Java Client applications different from client-server communication in HTML-based WebObjects applications. Understanding its details will help you write better designed, more advanced, and more secure Java Client applications.

This chapter covers the following topics as they relate to the distribution layer:

- business logic partitioning
- distribution layer objects and intralayer communication
- remote method invocations
- distribution channels
- distribution layer delegates

Business Logic Partitioning

In HTML-based WebObjects applications, all business logic (and the business objects that use that logic) lives on the server. Business objects are never sent to the client (the Web browser). Rather, selected data from those business objects is sent along with HTML user interface data.

In Java Client, however, business objects are sent to the client application for performance reasons. Java Client applications generally access more data than do distributed HTML applications. To limit the number of round trips to the server, copies of the business objects containing the data live on the client.

While this helps performance, it also presents security issues. In Java Client, business objects are Java objects, which can be decompiled and analyzed quite easily. So you never want to send sensitive business objects (objects containing private algorithms or data) to the client.

To control which business objects are sent to the client, you use *business logic partitioning*. As well as securing business data, business logic partitioning can also improve performance. The key to business logic partitioning is to minimize the amount of data sent from server to client while simultaneously minimizing the number of round trips over the network.

Design Recommendations

There are many ways to perform business logic partitioning. Often, you create a business logic class for the server and one for the client. These classes can be identical or their implementations can differ, depending on what data you want sent to the client.

Alternatively, you can create a common superclass from which the client and server subclasses inherit. In the common superclass, provide abstract declarations of the methods you want to be different in the two subclasses. In the client subclass, the methods should simply invoke remote methods of which concrete implementations exist in the server subclass.

For example, a common superclass might resemble this one:

```
package example.common;
import com.webobjects.eocontrol.*;
public abstract class Foo extends EOGenericRecord {
    public abstract String bar();
}
```

The client class (with a remote method invocation) would then resemble this class:

```
package example.client;
import com.webobjects.eocontrol.*;
public class Foo extends example.common.Foo {
    public String bar() {
     return (String) invokeRemoteMethod("clientSideRequestBar", null, null);
    }
}
```

The server-side class would then resemble this class:

```
package example.server;
import com.webobjects.eocontrol.*;
public class Foo extends example.common.Foo {
    public String bar() {
        return "secret string";
    }
    public String clientSideRequestBar() {
        return bar();
    }
}
```

The actual partitioning of your business logic begins in your EOModel. In EOModeler, you can assign custom classes to each entity in the model. See "Add Custom Business Logic" (page 106) in the advanced tutorial for an example.

Performance

As well as providing security for your business logic, partitioning can also confer performance improvements, depending on where computations take place. For example, if a particular computation requires a lot of data, and the client does not already have the data, that computation should occur on the server, as the server is closer to the data store.

Likewise, since Java Client requires rather robust clients, nonsensitive computations can occur on the client. This relieves the server from expending more cycles.

Remote Method Invocations

In Java Client applications, you may want some methods to execute only on the server for both security and performance reasons, such as when the method consumes a lot of system resources. Java Client defines two categories of remote method invocations: those that apply to business logic and those that apply to application logic.

Business Logic

If you partition your business logic in the recommended way, your client business logic classes shouldn't include any sensitive algorithms or computations. Rather, they should simply use remote method invocations to invoke concrete implementations of custom methods on the server that perform the sensitive computations. However, since remote method invocations require a round trip to the server, you *should* put nonsensitive algorithms in client-side business logic classes to reduce network traffic.

There are many methods defined throughout the Enterprise Object technology to perform remote method invocations. Client-side business logic classes that inherit from `com.webobjects.eocontrol.EOCustomObject` can use `invokeRemoteMethod` to invoke a method in the corresponding enterprise object on the server. The method takes three arguments: the method to invoke in the server-side class, a `java.lang.Class` object representing the argument types, and an object containing the arguments. Here's an example:

```
public void calculateRating() {
    invokeRemoteMethod("clientSideRequestCalculateRating", new Class[]
  {NSArray.class}, new Object[] {globalIDs});
}
```

This code invokes a method called `clientSideRequestCalculateRating` on the server, which takes an NSArray as an argument. You can pass `null` for both the second and third arguments if the remote method takes no arguments.

When you invoke a remote method on an enterprise object, the state of the client-side editing context is pushed to the server side. This guarantees that the business objects in the server-side computations are up to date with their client-side counterparts. Keep in mind that if you nest editing contexts on the client, all the editing contexts are pushed to the server side upon remote method invocation.

Note that `com.webobjects.eodistribution.client.EODistributedObjectStore` has remote method invocation methods (`invokeRemoteMethod` and `invokeRemoteMethodWithKeyPath`) that include a Boolean flag to control the pushing of the client-side editing context to the server. Setting this flag to `false` prevents the client from pushing its editing context state to the server. Since these methods are defined in EODistributedObjectStore, you must call them on an object store object if you invoke them from business logic classes.

Remote method invocations raise some security concerns since the client is assumed to be trusted. However WebObjects Java Client is well-prepared to handle these concerns. It includes built-in security features that prevent unauthorized remote method invocations. By default, remote method names must be prefixed with `clientSideRequest`, otherwise the EODistributionContext object on the server does not allow the remote method invocation. You can use delegates on the distribution context to implement your own security mechanisms for remote method invocations, as described in "Delegates" (page 100).

Note: This security contract applies only when the default distribution context delegate is used. That is, while the distribution context delegate is the Session object (which it is by default), remote method invocations that comply with the prefixing rule described in the above paragraph are allowed to execute.

However, if you set the distribution context delegate to an object other than Session, the prefixing contract does not apply and you must authorize each remote invocation explicitly by overriding these methods in the custom delegate class: `distributionContextShouldFollowKeyPath` (to authorize the distribution layer to follow the key path that references the custom delegate) and `distributionContextShouldAllowInvocation` (to allow the invocation of a method or methods in the custom delegate class).

Application Logic

Not all remote method invocations relate directly to business logic. Sometimes, you'd like to get information from the server that is specific to your application, but not particular to your application's business logic. This may include knowing what resources are available and how to handle user defaults.

Application-level remote methods are called with `invokeRemoteMethodWithKeyPath` and `invokeStatelessRemoteMethodWithKeyPath`, which are defined in EODistributedObjectStore. These methods are similar to `invokeRemoteMethod` except for two things. The receiver of the invocation can be any object (not just an enterprise object) that can be specified with a key path. The `keyPath` argument has special semantics:

- If `keyPath` is a fully qualified key path (for example, `session.editingContext`) the key path is followed starting from the invocation target of the EODistributionContext, which by default is the WOJavaClientComponent object.
- If `keyPath` is an empty string, the method is invoked on the WOComponent that is the invocation target of the EODistributionContext (typically a WOJavaClientComponent).
- If `keyPath` is `null`, the method is invoked on the server-side EODistributionContext.

The same security mechanism applies to these types of remote method invocations as to remote method invocations that occur on business logic classes (but only if the default distribution context delegate is used). That is, if the key path is `session`, the EODistributionContext on the server blocks all invocations sent with this method unless the `methodName` argument is prefixed with `clientSideRequest`.

If the key path specified is other than `session`, the EODistributionContext's delegate (on the server) must implement `distributionContextShouldAllowInvocation` and `distributionContextShouldFollowKeyPath` to allow the target method of the remote method invocation to be executed.

You can also invoke application-specific remote methods with `invokeStatelessRemoteMethodWithKeyPath`. Unlike `invokeRemoteMethodWithKeyPath`, it does not synchronize the client and server editing contexts. It is useful if you want to do something that has nothing to do with business logic, such as loading resources, running checks in background threads, and so on. It is much faster than `invokeRemoteMethodWithKeyPath` since it doesn't affect the object graph or editing contexts and avoids synchronization issues with client-side editing contexts in multithreaded applications.

In short, application-logic remote method invocations usually originate in custom Java Client controller classes, while business-logic remote method invocations usually originate from enterprise object classes (classes implementing the `com.webobjects.eocontrol.EOEnterpriseObject` interface).

Distributed Object Store

To perform remote method invocation on application logic, you invoke the methods on the client's distributed object store. The WebObjects API reference describes the distributed object store as follows:

"An EODistributedObjectStore functions as the parent object store on the client side of Java Client applications. 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."

You can get the distributed object store object with this code (assuming you haven't done anything special in the distribution layer with regard to the `EODistributedObjectStore`):

```
private EODistributedObjectStore _distributedObjectStore() {
    EOObjectStore objectStore = EOEditingContext.defaultParentObjectStore();
    if ((objectStore == null) || (!(objectStore instanceof EODistributedObjectStore)))
 {
        throw new IllegalStateException("Default parent object store needs to be an
          EODistributedObjectStore");
        }
        return (EODistributedObjectStore)objectStore;
}
```

Then you invoke the remote method on the object returned by the above method:

```
_distributedObjectStore().invokeRemoteMethodWithKeyPath(<arguments>);
```

Custom Code in Business Logic

There are a few things you need to know about using custom code in business logic classes. If you write methods that perform computations that require values in the enterprise object, two methods are provided to help you know when to invoke the custom computations: `awakeFromClientUpdate` and `prepareValuesForClient`.

The `awakeFromClientUpdate` method is invoked after the EOGenericRecord subclass on the server receives a notification that all the business objects have been received from the client. If you try to invoke a method from one of the class's `set` methods that performs a computation using values of attributes in your business logic, there is no guarantee that the server-side object has received all the values from the client you use in that calculation. However, if you invoke the method with said calculation in `awakeFromClientUpdate`, you are guaranteed to have all the business data from the client.

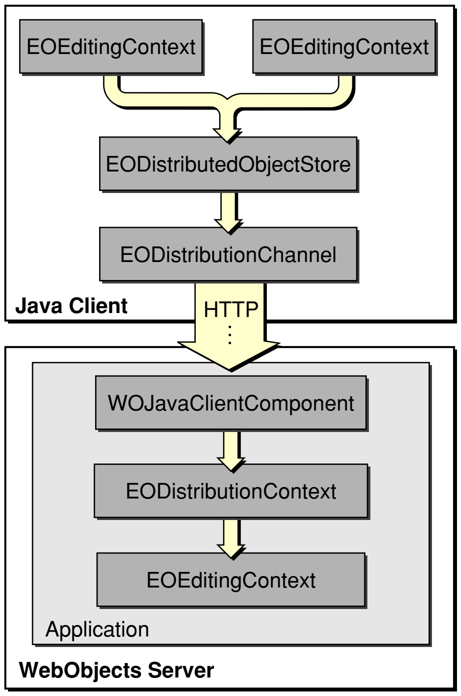
The `prepareValuesForClient` method is invoked in the EOGenericRecord subclass right before the business objects are sent back to the client (it is actually invoked right before the objects are encoded). You can override it to set a value before it is sent to the client if the value is only a client-side class property.

Distribution Layer Objects

The distribution layer's client-server communication mechanism relies on four objects: `com.webobjects.eodistribution.WOJavaClientComponent`, `com.webobjects.eodistribution.EODistributionContext`, `com.webobjects.eodistribution.client.EODistributedObjectStore`, and `com.webobjects.eodistribution.client.EODistributionChannel`.

The flow of information works like this: The client editing contexts talk to the EODistributedObjectStore (client side), which uses an EODistributionChannel to transfer objects across the network to the WOJavaClientComponent object, which uses an EODistributionContext to talk to the server-side editing context and to take care of generating responses to client requests. This flow is illustrated in Figure 5-1 (page 98).

Figure 5-1 Objects in the distribution layer



Let's examine each of these objects.

EODistributedObjectStore is the parent object store for all the editing contexts on the client. It makes the client editing contexts behave like a nested editing context to the server-side editing context. Its function is similar to that of the EODatabaseContext object, which lives on the server.

EODistributionChannel is responsible for sending data from the client to the server (it actually encodes the data).

The WOJavaClientComponent object is the target of the data sent by EODistributionChannel. It forwards data from the client's EODistributionChannel to the server's EODistributionContext. It is provided to isolate the application from different deployment environments and also plays a large role in application startup.

It is also the object that embeds Java Client in a WebObjects application.You can learn more about application startup by reading the API reference documentation for the distribution layer classes and the application-level classes in the application layer.

EODistributionContext has many functions: It keeps track of the state of the enterprise objects graph; it tracks which objects the client has fetched; and perhaps most importantly, it synchronizes business objects on the client and server applications.

Data Synchronization

The distribution layer is responsible for synchronizing the client and server object stores. The data flow in a Java Client application occurs like this:

1. The user makes a query and the fetch specification is forwarded by the client's EODistribution layer to the server's EODistribution layer.
2. The normal WebObjects mechanisms take over, and a SQL call is eventually made to the database server.
3. The database server returns rows of requested data; these rows are mapped to enterprise object instances.
4. The server's EODistribution layer sends copies of the requested data to the client.
5. The client's EODistribution layer receives the objects and registers them with the client's editing context (the data is cached in the client's object graph).
6. Through the client's display group and association mechanisms, the user interface is populated.

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 object graph is pushed to the server. If the business logic on the server validates these changes, the changes are committed to the database.

Synchronization of the client and server's object graphs occurs automatically: Java Client automatically pushes updates from the server to the client.

Note: 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 object graphs on the client are usually a subset of those on the server. 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.

Using SSL

The distribution channel in Java Client (EODistributionChannel) distributes data between the client and server applications. By default, Java Client uses HTTP as the transport mechanism (EOHTTPChannel), but you can subclass EOHTTPChannel to provide a custom mechanism such as SSL. Assuming that you're using Sun's JSSE SSL classes, you can use the class in Listing 5-1.

Listing 5-1 Subclass EOHTTPChannel to use SSL

```
package com.mycompany.client;

import java.io.*;
import java.net.*;
import java.security.*;

public class SSLChannel extends com.webobjects.eodistribution.client.EOHTTPChannel {

    static {
        System.setProperty("java.protocol.handler.pkgs",
          "com.sun.net.ssl.internal.www.protocol");
        Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider());
    }

    public Socket createSocket(String protocol, String hostName, int portNumber) throws
          IOException {
        if (!"https".equals(protocol)) {
            return super.createSocket(protocol, hostName, portNumber);
        }

        Socket socket =
           javax.net.ssl.SSLSocketFactory.getDefault().createSocket(hostName, 443);
        socket.setTcpNoDelay(true);
        return(socket);
    }
}
```

Make sure to add this subclass to the Web Server (client application) target in your project.

After you add this class, you need to add a binding on the server-side distribution layer component to tell the application to use the custom subclass. Add a binding to the `JavaClient.wo` component called `distributionChannelClassName`. Its value is the fully qualified name of the distribution layer subclass that uses SSL. Using the subclass in Listing 5-1, this is `com.mycompany.client.SSLChannel`.

If you've already configured your Web server to use SSL, there are no more steps to perform to use SSL in Java Client applications. Unfortunately, configuring a Web server to use SSL, obtaining SSL certificates, and other tasks necessary to implement a secure protocol are rather complicated tasks and are beyond the scope of this document.

However, if you're deploying on Mac OS X, the included Apache Web server includes the mod_ssl module and documentation for configuring SSL. See `file:///Library/Documentation/Services/apache_mod_ssl/ssl_overview.html` on a Mac OS X system.

Delegates

Since the Java Client distribution layer is rather complex, it provides a number of delegates you can use to customize its behavior, which saves you from subclassing. You can set delegates for EODistributionContext and EODistributionChannel to

- change the security mechanism for validating remote method invocations

- implement a custom encryption and decryption scheme for data transfer over the network
- control access to business objects
- handle client-side I/O exceptions
- handle server-side exceptions such as validation, null pointer exceptions, and session timeouts

The default delegate of the distribution layer is the application's Session object. This is set in the EODistributionContext's constructor and you set custom delegates with the `setDelegate` method in EODistributionChannel and EODistributionContext, as described in "Setting the Delegate" (page 101). Because the Session object is the default delegate, you can override the methods that are defined in the EODistributionContext.Delegate class in the Session class and they will be found and invoked by the distribution layer automatically.

To secure access to the data sources with which the server-side application communicates, you override methods defined in the class EODistributionContext.Delegate. A common design pattern is to override all the methods in the delegate class to return `true` only if a user has authenticated. The easiest and most common way to do this is to store the state of a user's login in the Session object, as is described in "Building a Login Window" (page 251) and as shown in the JCDiscussionBoard example project.

Assuming that you store a user's authenticated state in an instance variable (`isAuthenticated`) in the Session class, you can use the code in Listing 5-2 to prevent unauthenticated access to the data store. By restricting access to the class description of entities in the application, you restrict access to and editing of all entities in the application.

Listing 5-2 Use a delegate to restrict access to data

```
public boolean distributionContextShouldAllowAccessToClassDescription(
     EODistributionContext context, EOClassDescription classDescription) {
     return (isAuthenticated());}
```

Other delegates defined in EODistributionContext.Delegate allow you to control whether editing contexts are allowed to save, whether particular fetch specifications are allowed to be invoked, and whether the target methods of remote method invocations are allowed to be invoked. There are also two methods you can use to encrypt and decrypt data across the wire, though using SSL as the EODistributionChannel protocol is probably a better idea to secure client-server communications. See the API reference for EODistributionContext.Delegate for more details.

Setting the Delegate

To set the distribution context's delegate, you need to know when the distribution context has been instantiated. You can use the code in Listing 5-3 to listen for the notification that is sent immediately after the distribution context is instantiated and to invoke custom behavior after that event has occurred.

Listing 5-3 Set the distribution context's delegate

```
import com.webobjects.foundation.*;
import com.webobjects.appserver.*;
import com.webobjects.eocontrol.*;
import com.webobjects.eodistribution.*;

public class Session extends WOSession {
    public EODistributionContext dContext = null;
    public MyDistributionDelegate myDelegate = null;

    public Session() {
        super();

        myDelegate = new MyDistributionDelegate();

        NSNotificationCenter.defaultCenter().addObserver(this,
            new NSSelector("listenForContext", new Class[] { NSNotification.class } ),
            EODistributionContext.DistributionContextInstantiatedNotification, null);
    }

    public void listenForContext(NSNotification notification) {
        dContext = (EODistributionContext)notification.object();
        dContext.setDelegate(myDelegate);
    }
}
```

In Session's constructor in Listing 5-3, the class registers with the default notification center to listen for the `EODistributionContext.DistributionContextInstantiatedNotification` notification. When that notification is received from the notification center, the method specified with the NSSelector argument (the method named `listenForContext`, which takes a single NSNotification object as an argument) is invoked. The `listenForContext` method is passed a reference to the distribution context object on which it invokes `setDelegate`. It sets the delegate to a custom class called MyDistributionDelegate, which is instantiated in Session's constructor.

To set the delegate for the distribution channel object, simply replace EODistributionContext in Listing 5-3 with EODistributionChannel.

Enhancing the Application

In this chapter you'll further customize the application you created in the basic tutorial. You'll learn how to

- add custom business logic to your application
- use NSValidation to validate data
- use remote method invocations
- subclass controller classes to customize applications
- use rules to change application behavior
- add custom actions to the client application

Customization Techniques

This tutorial uses some of the Direct to Java Client customization techniques. Before teaching you how to implement them, however, this section provides a summary of all the customization techniques available in Direct to Java Client, including their costs and appropriate usage.

Table 6-1 (page 103) compares the five customization techniques using several criteria.

Table 6-1 Consequences of each customization technique

	Synchronization with data model	**Maintainability**	**Source code writing**	**Localization**
Assistant	Mostly automatic	Easy	None	Easy
Custom rules	Easy	Easy	None	Easy
Freezing XML	More difficult	Moderate	Minimal	More difficult
Freezing interface files	More difficult	Moderate to difficult	Minimal	Moderately easy, using rule system
Custom controller classes	Not applicable	Difficult	Much	Easy, using EOUserDefaults

The first customization tool is the Direct to Java Client Assistant, which you've already used in "Building a Simple Application" (page 47) It allows you to

- change an entity's type (main, enumeration, or other)
- change the properties that are displayed in any of the four tasks (form, query, list, and identify)
- add new property keys

- change the widget type of property keys
- make basic customizations to the client application, such as changing the window titles and setting window sizes

The costs of using Assistant are very low: If you make changes to your data model, in most cases the rule system picks them up. (Some changes you make in Assistant, such as changing entity types, may not guarantee that changes in your model are picked up by the rule system.) You should do as much customization as possible within Assistant before moving on to more advanced customization techniques, which make synchronizing the user interface with the data model more complicated.

The second customization tool is writing custom rules. You do this in the Rule Editor application. The look and behavior of Direct to Java Client applications is defined by rules that work with the WebObjects rule system. The rule system is an integral part of the two WebObjects rapid development solutions, Direct to Web and Direct to Java Client. You can learn more about it in "Inside the Rule System" (page 163)

Using custom rules is more difficult than just using Assistant, but the costs of using the rules are no higher than using Assistant (Assistant simply writes rules based on the customizations you make within it.) Many custom rules apply to specific entities, so if you change the entities in your model, you may invalidate some rules. But this is easily fixed by changing the argument in the rule that references a particular entity.

A simple rule is to specify the minimum width for all windows in an application:

Left-Hand Side:	`(controllerType='windowController')`
Key:	`minimumWidth`
Value:	`512`
Priority:	`50`

You can define this characteristic for windows throughout your application programmatically, but it's much easier and more maintainable to just write a rule. Rules are very abstract, and once you learn their syntax and semantics, you'll find them to be a powerful customization technique.

The next customization technique is freezing XML, which allows you to explicitly state the result of a rule system request. The dynamically generated user interfaces Direct to Java Client produces are described in XML. In Assistant, the XML pane shows the XML description for each task for each entity for each window type in your application. Usually you start with this generated XML and customize it to suit your needs. This technique is fully explained in the chapter "Freezing XML User Interfaces" (page 199).

Freezing XML incurs more costs than writing custom rules or using Assistant since the user interface description is static. If you make changes to your data model, you'll have to manually find and update any specific references to the entities and attributes in the user interface description. Since the XML descriptions are very abstract, this task is not too difficult. But, you should use Assistant as much as possible to customize your application before moving on to frozen XML.

In addition to using frozen XML, you can use frozen interface files created in Interface Builder. Although this gives you more control over the user interface, it makes maintenance more difficult, it makes platform-specific layout and localization much harder, and it makes data model synchronization more challenging. "Mixing Static and Dynamic User Interfaces" (page 209) teaches you to how freeze interface files and integrate them in dynamically generated user interfaces.

Among the most advanced techniques is writing custom controller classes. These are usually subclasses of EOController, and they can include any Swing component or any component written in Java. For instance, if you'd like a `JPasswordField` widget somewhere in your application, you'd have to write a custom controller class since this widget isn't provided for you by default. Then, in the XML description for the window or modal dialog, you'd specify the custom controller class using the `className` attribute.

Using custom controller classes provides you with total control over the user interface, but it incurs high costs. It requires you to write source code (an inherently buggy process), which makes data model synchronization quite difficult, especially if you use the custom controller class with frozen XML.

Enhance the EOModel

The application in the basic tutorial uses a rather simple data model that offers little opportunity to customize applications that use it. A more advanced model will better demonstrate the customization features of Direct to Java Client. Since you'll be modifying the model, however, it's kept rather simple so you won't have to spend too much time editing it.

1. Open the `Admissions.eomodeld` file from within the Admissions project.

2. Add these attributes to the Student entity:

   - Name: `act`; Column: ACT; External Type: `int`; Internal Data Type: `Integer`.
   - Name: `sat`; Column: SAT; External Type: `int`; Internal Data Type: `Integer`.
   - Name: `firstContact`; Column: FIRST_CONTACT; External Type: `date`; Internal Data Type: `Date`. *Don't lock on this attribute: Deselect the lock icon in the attribute's row.*

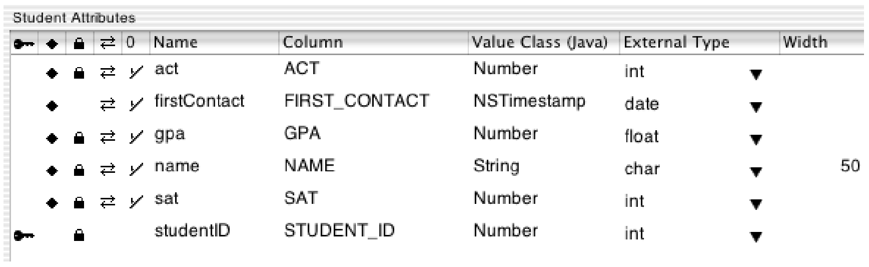
   Make all the new attributes client-side class properties. By default, they should also be set as server-side class properties, so make sure the diamond icon is present for all the new attributes.

3. Since you added attributes to the entity, you must synchronize the model and the database schema. Refer to "Using the Application" (page 71) and Figure 3-21 (page 72) for a reminder.

> **Note:** You may receive an exception when synchronizing the schema in this step. Schema synchronization can be a useful feature but you probably don't want to use it on a production database that holds important data.

The Student entity should now resemble Figure 6-1 (page 105).

Figure 6-1 The updated Student entity



Student Attributes

🔑	◆	🔒	⇄	0	Name	Column	Value Class (Java)	External Type		Width
	◆	🔒	⇄	✓	act	ACT	Number	int	▼	
	◆		⇄	✓	firstContact	FIRST_CONTACT	NSTimestamp	date	▼	
	◆	🔒	⇄	✓	gpa	GPA	Number	float	▼	
	◆	🔒	⇄	✓	name	NAME	String	char	▼	50
	◆	🔒	⇄	✓	sat	SAT	Number	int	▼	
🔑		🔒			studentID	STUDENT_ID	Number	int	▼	

Add Custom Business Logic

As the basic tutorial illustrates, you can go far in developing a Direct to Java Client application without writing any code. However, the real power of a Java Client application is in the enterprise objects you create and customize. The behavior or business logic you add to enterprise objects brings stored data to life.

By default, EOModeler assigns new entities the class EOGenericRecord. EOGenericRecord 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 objects class. This class includes the default behavior provided by EOGenericRecord as well as the custom behavior you implement.

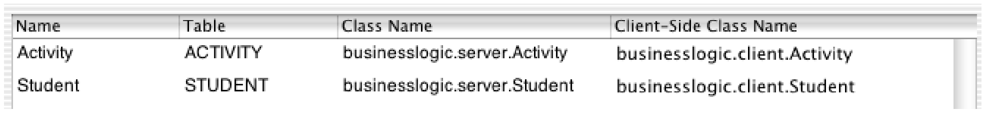
To use custom business logic in your application, you assign custom classes to the entities in your model.

1. In EOModeler, select the Admissions 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 Class Name cell for Student in the table and enter `businesslogic.server.Student`.

3. Double-click the Client-Side Class Name cell for Student and change `server` to `client` so it reads `businesslogic.client.Student`.

4. Repeat these steps for the Activity entity, substituting `Activity` for `Student` in the package name.



Name	Table	Class Name	Client-Side Class Name
Activity	ACTIVITY	businesslogic.server.Activity	businesslogic.client.Activity
Student	STUDENT	businesslogic.server.Student	businesslogic.client.Student

5. Save the model.

The recommended naming convention of custom class names is to adhere to Java package syntax.

By giving both the Class Name (server) attribute and the Client-Side Class Name (client) attribute custom class names, you are telling the model to use custom classes on both the client and the server. But this isn't required—you can implement a class only on the server or only the client, depending on your needs. See "Design Recommendations" (page 93) for more information.

Once you specify a custom class for an entity in EOModeler, you can generate Java source files for that entity. Before doing that should prepare your project to handle the new files.

Prepare the Project for Custom Logic

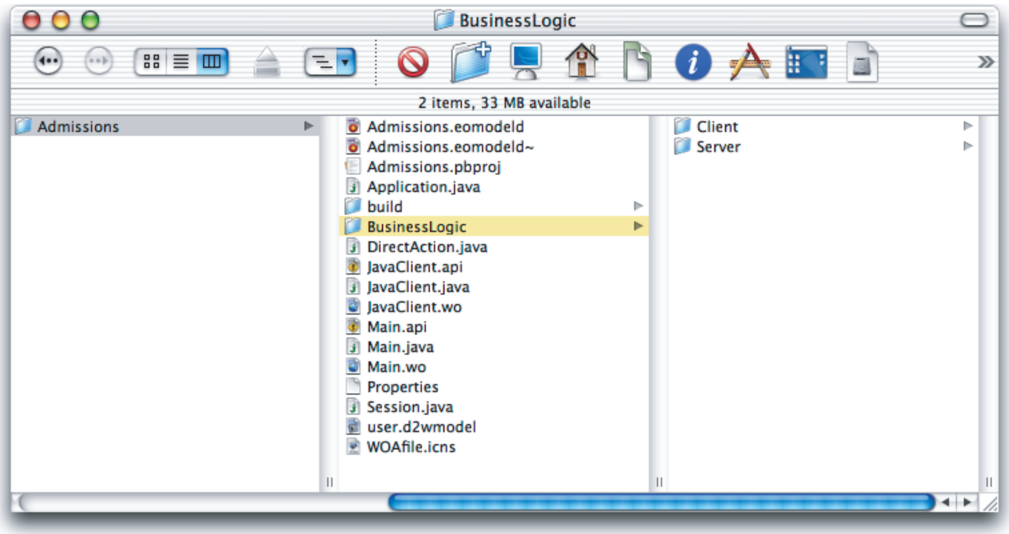
Project Builder stores most of a project's files at the top level of the project directory in the file system even though it organizes files in logical groupings inside the project itself. It's a good idea to separate your business logic files from other WebObjects files both in the project directory in the file system and in logical groupings inside the Project Builder project.

Follow this step to create a `BusinessLogic` directory with subdirectories in the file system, and to create a `BusinessLogic` group in the project:

Create the following directories at the top level of your project directory (do this in the file system, not in Project Builder): `BusinessLogicBusinessLogic/ClientBusinessLogic/Server`

The directory structure should look like Figure 6-2 (page 107).

Figure 6-2 Directory structure for custom business logic



Generate Source Files

EOModeler can generate Java files for your model. You'll use these source files to add custom business logic to your enterprise objects.

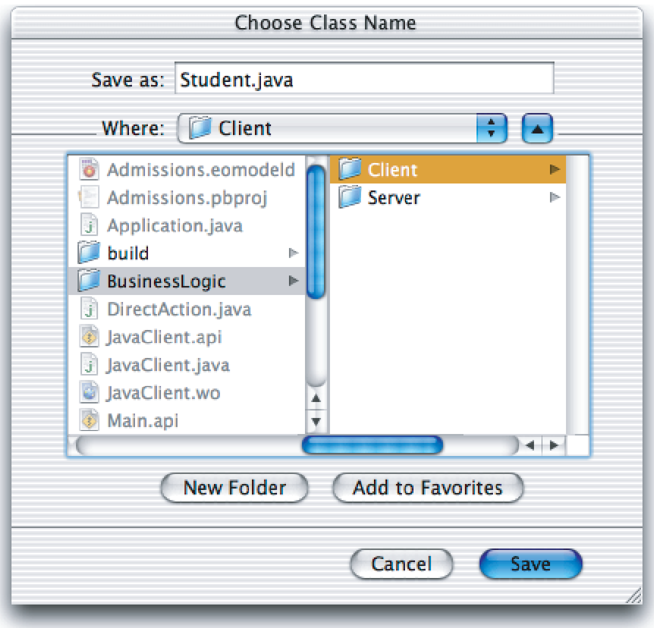
Note: In WebObjects 5.1 with certain versions of Mac OS X and the developer tools, EOModeler does not prompt you for a location for the class files it generates. Rather, it attempts to save files in the model's directory. To work around this bug, you'll have to manually move the generated class files to the correct directories. The bug is fixed in WebObjects 5.2.

Follow these steps to generate Java files for the client:

1. In EOModeler, select the Student entity.
2. Choose Property > Generate Client Java Files.
3. Select the `Client` directory inside the `BusinessLogic` directory in the project, as shown in Figure 6-3 (page 108).
4. Click Save.

5. Repeat the process for the Activity entity.

Figure 6-3 Save Client Java files in BusinessLogic/Client



Follow these steps to generate Java files for the server:

1. In EOModeler, select the Student entity.
2. Choose Property > Generate Java Files.
3. Select the `Server` directory inside the `BusinessLogic` directory in the project.
4. Click Save.
5. Repeat the process for the Activity entity.

The Java class files generated by EOModeler include the necessary import declarations as well as constructors and accessor methods derived from the properties of the entity defined in the model file.

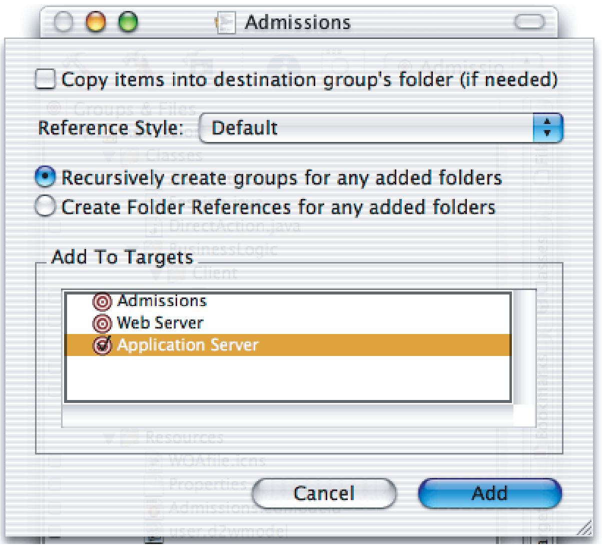
Although you told EOModeler where to put the generated files, Project Builder did not automatically add them to the project.

Follow these steps to import the generated files into Project Builder:

1. Select the Classes group in the Files list of Project Builder (click the Files tab if this list isn't visible).
2. Choose Project > Add Files.
3. Select the `BusinessLogic` directory and click Open. This creates a new group and imports the `BusinessLogic` directory and its subdirectories into the group.

4. Select Application Server as the target as shown in Figure 6-4. Also make sure that "Recursively create groups for any added folders" is selected.

Figure 6-4 Import BusinessLogic directory


Admissions
Copy items into destination group's folder (if needed)
Reference Style: Default
Recursively create groups for any added folders
Create Folder References for any added folders
Add To Targets
Admissions
Web Server
Application Server
Cancel
Add

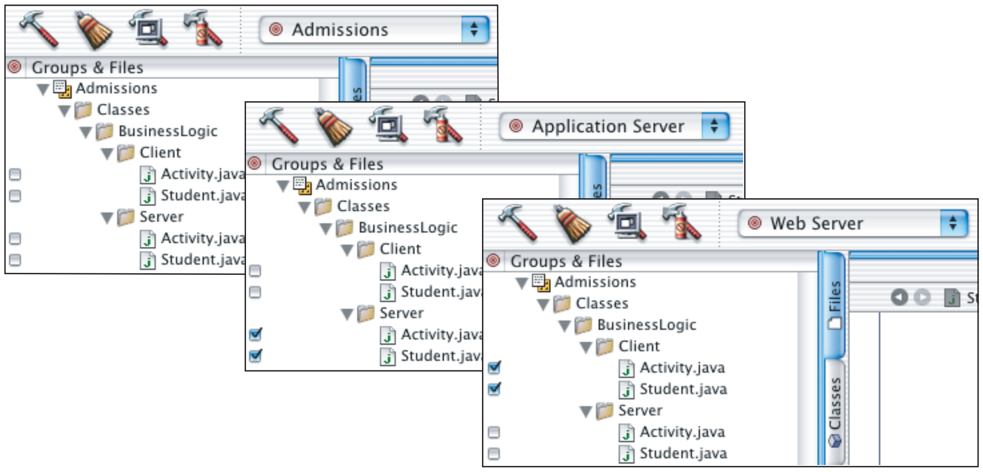
5. Click Add. The new files should appear in the Files list as illustrated in Figure 6-5.

6. After the import, change the target for the files in `BusinessLogic/Client` to Web Server. Do this by changing the target to Web Server in the target's pop-up menu and selecting the checkbox to the left of each file in the `BusinessLogic/Client` group. Refer to Figure 6-5 for clarity.

   Make sure you also disassociate the files in `BusinessLogic/Client` from the Application Server target by switching to that target and deselecting the checkbox to the left of each file in that group. The client Java files must be built as part of the Web Server target rather than as part of the Application Server target.

   Make sure that Admissions is the target selected in the targets pop-up menu after you've correctly associated the imported files with their targets.

Figure 6-5 BusinessLogic group with imported files and associated targets



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

If you examine the code in any of the imported classes, you'll notice that the class generated by EOModeler does not have actual instance variables or fields. Rather, the methods to access the attributes of the custom enterprise objects are implemented using key-value coding.

Behind the Steps

Step 4: You were instructed to assign all the imported classes to the Application Server target for convenience. You could also import the files in `BusinesssLogic/Client` and `BusinessLogic/Server` separately and assign them to the correct target at that time.

Step 6: As an alternative to importing all the custom Java classes at once and then changing the target accordingly, you can also import the server and client classes separately and assign them to the appropriate target at that time.

Prepare Application for Business Logic

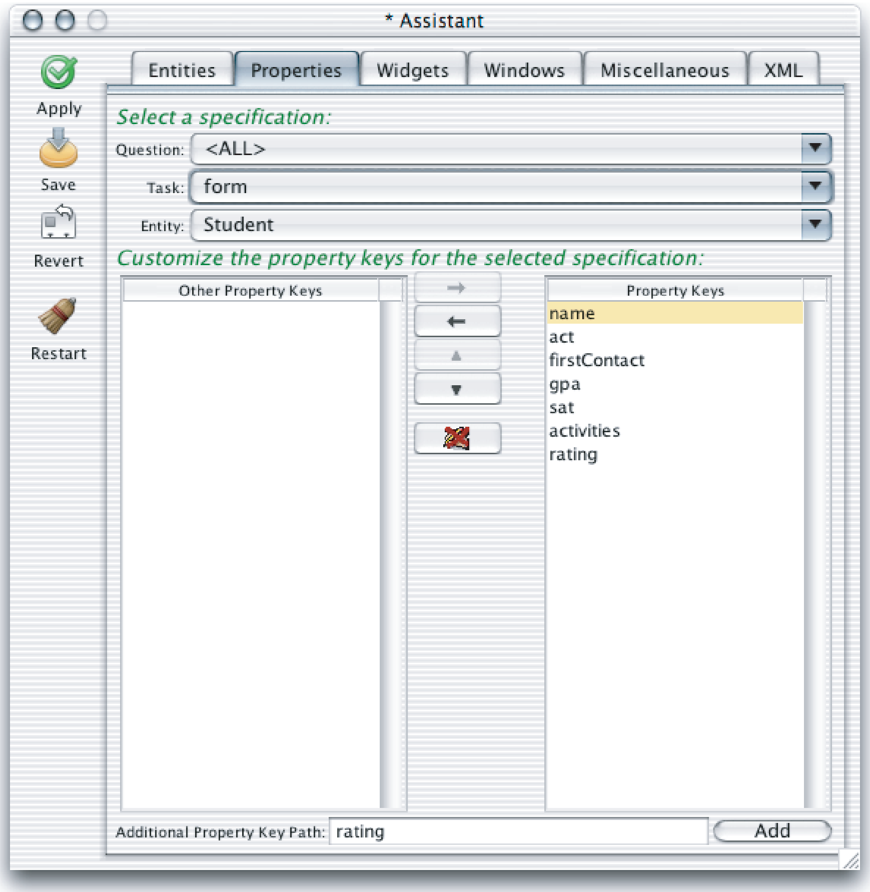
The business logic you'll add is quite simple: It calculates a rating for a student by aggregating the three scores in the database: ACT, SAT, and GPA. You can use Assistant to prepare the application for this new business logic.

Here's how:

1. Build and run the application and open Assistant. You have to build the application again since you changed the model.

2. In the Properties pane, add a new property key path called `rating` for Task > form and Entity > Student using the Additional Property Key Path text field and the Add button as shown in Figure 6-6 (page 111). See "Additional Property Key Path Field" (page 89) for more information on what's happening in this step.

Figure 6-6 Add a property key for the form task



Select a specification:

Question: <ALL>

Task: form

Entity: Student

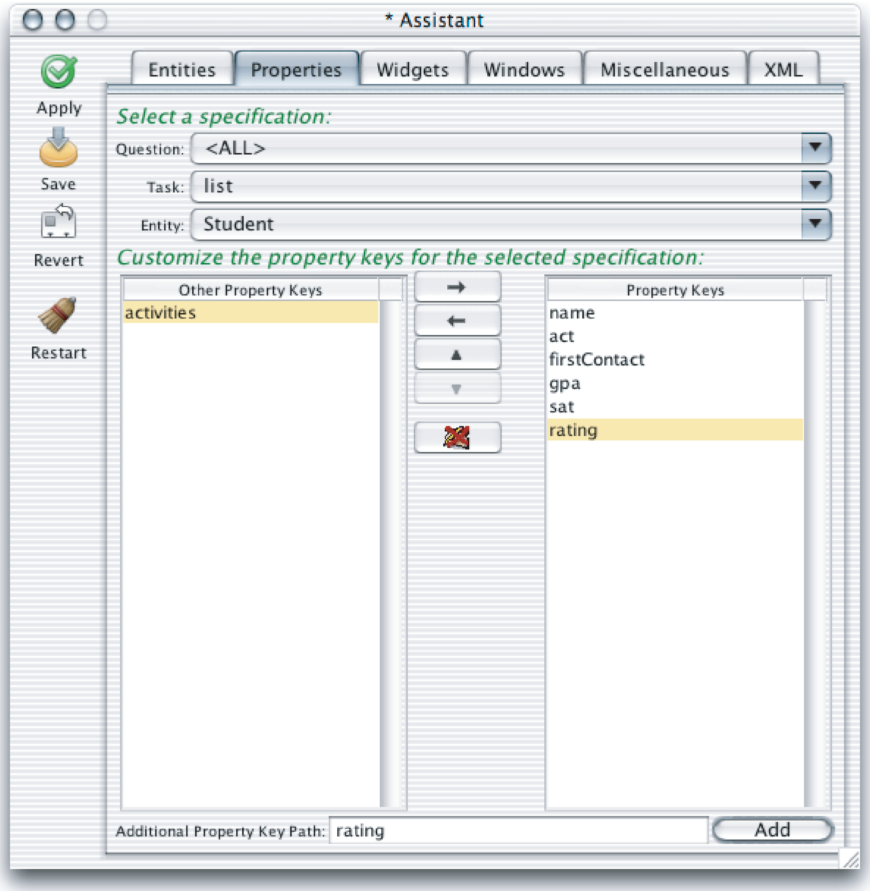
Customize the property keys for the selected specification:

Property Keys
name
act
firstContact
gpa
sat
activities
rating

Additional Property Key Path: rating

3. Since you'd like to see the rating displayed in the list view of a query window, you also need to add the additional property for the list task. Choose list in the Task pop-up menu and click Add, as shown in Figure 6-7.

Figure 6-7 Additional property key for list task



Select a specification:
Question: <ALL>
Task: list
Entity: Student
Customize the property keys for the selected specification:

Other Property Keys
activities

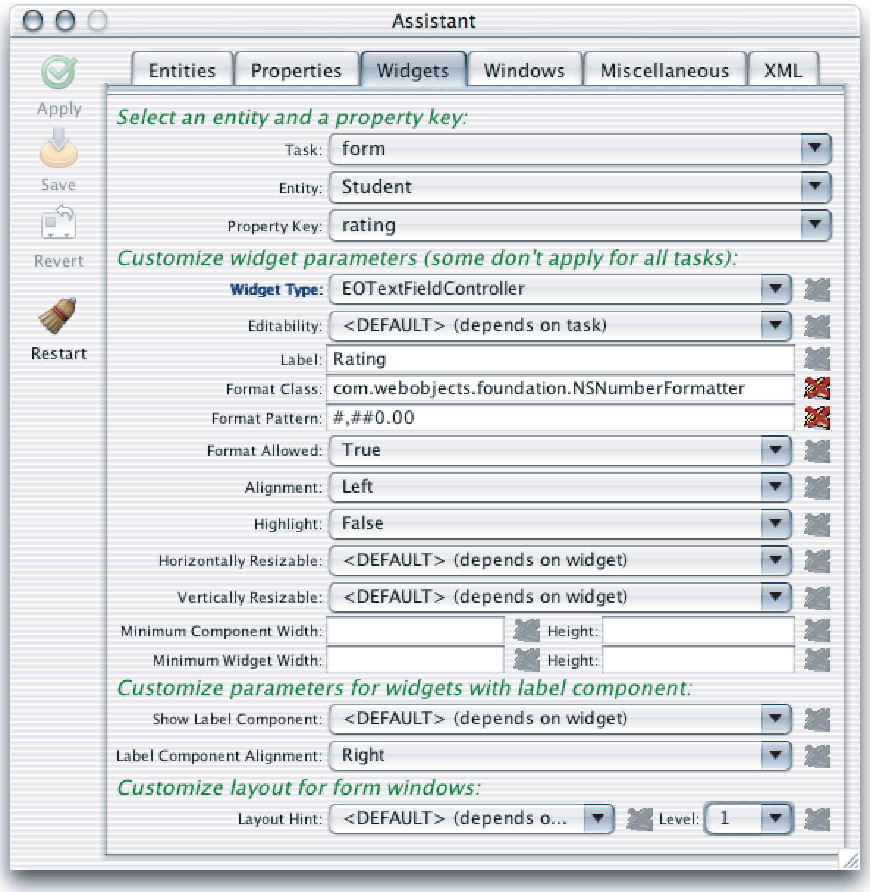
Property Keys
name
act
firstContact
gpa
sat
rating

Additional Property Key Path: rating

4. The new property will be associated (via an EOAssociation, see "Associations" (page 43)) with a method of the same name in a client-side business logic class for the entity (`businesslogic.client.Student` in this case).

To make this association, switch to the Widgets pane and choose Task > form and Entity > Student, then select rating in the Property Keys list. From the Widget Type pop-up menu, choose EOTextFieldController if it is not already chosen. Doing this binds the association aspect of the EOTextFieldController widget (rating) with the `rating` method, which you'll define in a few steps.

Figure 6-8 Change the widget type to make the association.



Select an entity and a property key:

Task: form
Entity: Student
Property Key: rating

Customize widget parameters (some don't apply for all tasks):

Widget Type: EOTextFieldController
Editability: <DEFAULT> (depends on task)
Label: Rating
Format Class: com.webobjects.foundation.NSNumberFormatter
Format Pattern: #,##0.00
Format Allowed: True
Alignment: Left
Highlight: False
Horizontally Resizable: <DEFAULT> (depends on widget)
Vertically Resizable: <DEFAULT> (depends on widget)
Minimum Component Width: Height:
Minimum Widget Width: Height:

Customize parameters for widgets with label component:

Show Label Component: <DEFAULT> (depends on widget)
Label Component Alignment: Right

Customize layout for form windows:

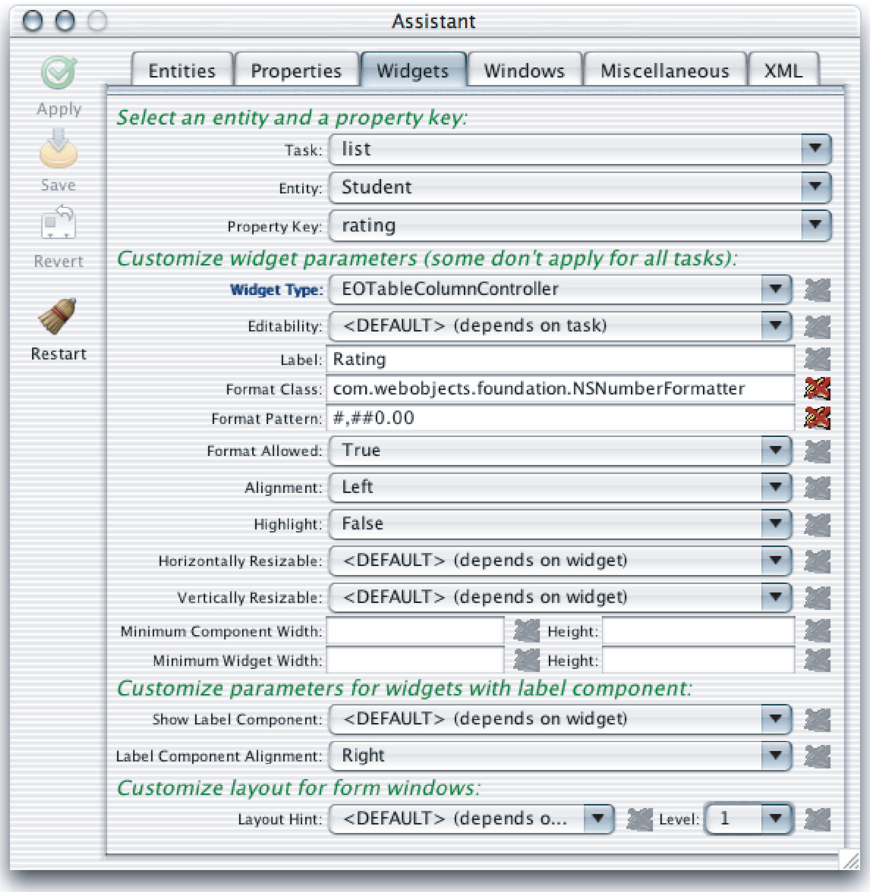
Layout Hint: <DEFAULT> (depends o... Level: 1

5. Since the rating is calculated on the server side, the text field should be marked as not editable by the user. So, while in the Widgets pane, select Never in the Editability pop-up menu.

6. Finally, you should apply a number formatter to the widget so the number displayed is more meaningful. Change the Format Class field to read `com.webobjects.foundation.NSNumberFormatter`. Formatters need a pattern, and since the rating is a decimal number, the Format Pattern field should be `#,##0.00` as shown in Figure 6-8 (page 113). See the API reference documentation for NSNumberFormatter for more information on format patterns.

7. Since the rating also appears as a column in list views, switch the task to list and set the format options for the EOTableColumnController as shown in Figure 6-9 (page 114).

Figure 6-9 Change formatter for property in list view



Select an entity and a property key:

Task: list
Entity: Student
Property Key: rating

Customize widget parameters (some don't apply for all tasks):

Widget Type: EOTableColumnController
Editability: <DEFAULT> (depends on task)
Label: Rating
Format Class: com.webobjects.foundation.NSNumberFormatter
Format Pattern: #,##0.00
Format Allowed: True
Alignment: Left
Highlight: False
Horizontally Resizable: <DEFAULT> (depends on widget)
Vertically Resizable: <DEFAULT> (depends on widget)
Minimum Component Width: Height:
Minimum Widget Width: Height:

Customize parameters for widgets with label component:

Show Label Component: <DEFAULT> (depends on widget)
Label Component Alignment: Right

Customize layout for form windows:

Layout Hint: <DEFAULT> (depends o... Level: 1

8. Save changes and quit the client and server applications.

Add Custom Code

You now need to add a method for the new property you added in Assistant. The new `rating` attribute in the Student entity is designed to aggregate ACT and SAT scores and GPAs into a numeric rating based on how each of those attributes is weighted. You need to add a method to perform the calculation, a method to invoke the calculation, and class constants to define the weighting.

The algorithm used to calculate the rating is “sensitive” business logic, so it should exist only on the server side. The client business logic class simply invokes the concrete implementations of the rating methods on the server side.

Add these class constants to the server-side `Student.java` file:

```
private static final double ACT_WEIGHT = 0.30;
private static final double SAT_WEIGHT = 0.30;
private static final double GPA_WEIGHT = 0.40;
```

Add this method to the server-side `Student.java` file:

```
public Number rating() {
    float aggregate = 0;
    float satTemp;
    float actTemp;
    float gpaTemp;

    if (sat() != null && act() != null && gpa() !=null) {
        satTemp = sat().floatValue() / 1600;
        actTemp = act().floatValue() / 36;
        gpaTemp = gpa().floatValue() / 4;

        aggregate = (float)(((gpaTemp * GPA_WEIGHT) + (actTemp * ACT_WEIGHT)
          + (satTemp * SAT_WEIGHT)) * 10);
    }

    return (new Float(aggregate));
 }
```

Add a method called `clientSideRequestRating` in the server-side `Student.java` file that invokes the `rating` method, as shown:

```
public Number clientSideRequestRating() {
        return rating();
}
```

Add this code to client-side `Student.java` file to invoke the remote method:

```
 public Number rating() {
        return (Number)(invokeRemoteMethod("clientSideRequestRating", null,
              null));
 }
```

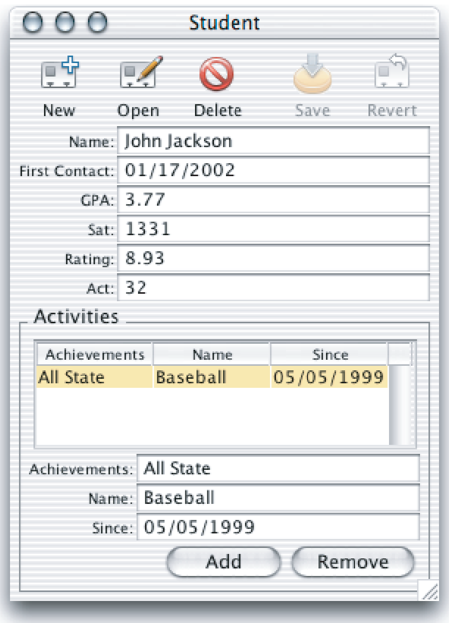
In the last section, you bound the association aspect of the EOTextFieldController (rating) to a method called `rating` in the client-side business logic class. You've just defined this method, so now whenever the rating property needs a value, the `rating` method is invoked. It's that easy—Java Client handles all the communication between the business logic and the user interface for you.

There is more going on behind the scenes, though. The `rating` in the client-side business logic class invokes a remote method called `clientSideRequestRating` in the server-side business logic class. This method in turn invokes a method called `rating`, which actually performs the calculation.

Rebuild and run the application. Make a new student record and see how the rating field is populated upon saving as shown in Figure 6-10 (page 116). That figure is shown with a layout hint of Row (see "Widgets Pane" (page 90) for more information).

Note: Whenever `rating` is requested, a round trip to the server is made to perform the remote method invocation. To lessen network traffic, you should consider caching the value in the client-side enterprise object.

Figure 6-10 The rating field in action



Name: John Jackson
First Contact: 01/17/2002
GPA: 3.77
Sat: 1331
Rating: 8.93
Act: 32

Activities

Achievements	Name	Since
All State	Baseball	05/05/1999

Achievements: All State
Name: Baseball
Since: 05/05/1999

To learn how to implement continuous change notification for the rating field, see "Continuous Change Notification" (page 197).

Validation

WebObjects provides some useful classes and methods to validate user input. You should validate the entered data for each of the three score fields. To do this, add the following code in the server-side `Student.java` class:

```
public Number validateSat(Number score) throws NSValidation.ValidationException {
    if ((score.intValue() > 1600) || (score.intValue() < 0)) {
        throw new NSValidation.ValidationException("Invalid SAT score");
    }
    else
        return score;
}

public Number validateAct(Number score) throws NSValidation.ValidationException {
    if ((score.intValue() > 36) || (score.intValue() < 0)) {
        throw new NSValidation.ValidationException("Invalid ACT score");
    }
    else
        return score;
}
```

```
public Number validateGpa(Number score) throws NSValidation.ValidationException {
    if ((score.floatValue() > 4.0) || (score.floatValue() < 0.0)) {
        throw new NSValidation.ValidationException("Invalid GPA");
    }
    else
        return score;
}
```

The code you added is rather trivial, but it demonstrates a particularly powerful feature of WebObjects—validation. The NSValidation class in the Foundation framework provides this functionality. By throwing a `NSValidation.ValidationException`, a method tells Enterprise Objects that the current object graph is not cleared to be saved to the database.

In this case, if one of the attributes fails to validate, the object graph is not cleared by NSValidation and the current record won't be committed to the data store until a valid value is entered.

You were instructed to put all the validation methods in the server-side business logic class, but this is not necessary. In fact, it often makes more sense to validate some values on the client. This reduces network traffic (there is no round-trip to the server to perform the validation) and increases overall application performance. Experiment with this by moving one of the validation methods to the client-side business logic class.

Validation methods are of the form `validate`*Attribute*. In this example, be sure that `validateGpa` is capitalized correctly—`validateGPA` will not invoke validation on the `gpa` attribute.

If you write validation methods, they are invoked in the framework by various classes and interfaces such as EOValidation, EODisplayGroup, and EOEditingContext. Validation is performed for these activities:

- updating the client-side distribution context (`validateForUpdate`)
- saving to the database (`validateForSave`)
- deleting from the database (`validateForDelete`)
- inserting a new record (`validateForInsert`)
- updating the server-side database context (`validateForUpdate`)

Initial Values

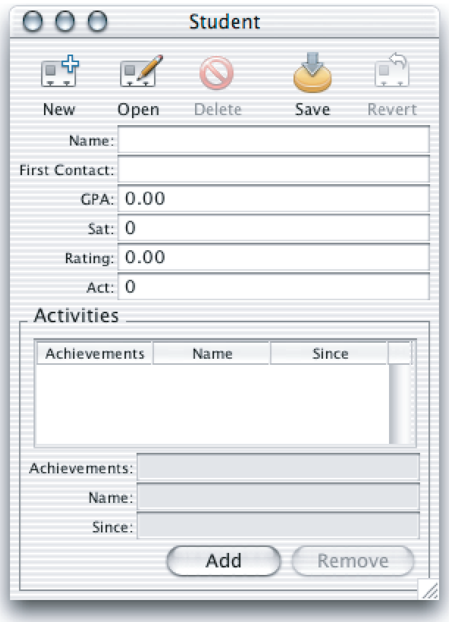
When you create a new record, you might want to supply some default values for the fields in that record. Although none of the fields in the Student record really need a default value, you'll override `awakeFromInsertion` in order to learn how to give a field a default value.

Add this code in the server-side `Student.java` file:

```
public void awakeFromInsertion(EOEditingContext context) {
        super.awakeFromInsertion(context);
        if (gpa() == null) {
            setGpa(new BigDecimal("0"));
        }
        if (sat() == null) {
            setSat(new BigDecimal("0"));
        }
        if (act() == null) {
            setAct(new BigDecimal("0"));
        }
        if (name() == null) {
            setName("");
        }
}
```

Build and run the application and create a new student record. You'll notice that some of the fields are populated in the new record as shown in Figure 6-11.

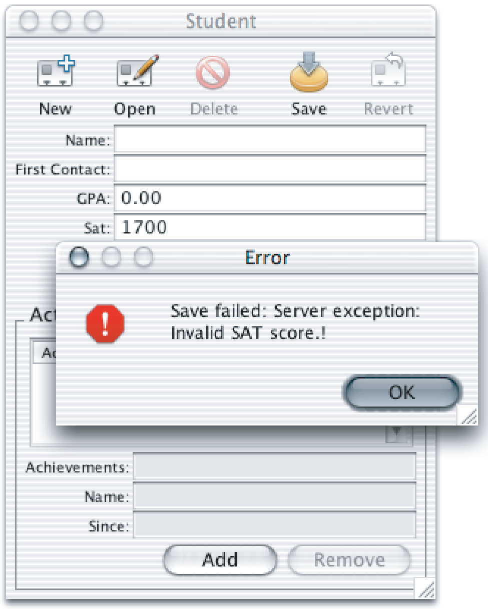
Figure 6-11 Initial values



Name:
First Contact:
GPA: 0.00
Sat: 0
Rating: 0.00
Act: 0
Achievements:
Name:
Since:

Also try entering some invalid data to see how the validation you implemented works. If you enter an invalid score, you should get a validation exception message when saving, as shown in Figure 6-12.

Figure 6-12 Validation exception message



Name:
First Contact:
GPA: 0.00
Sat: 1700

Save failed: Server exception:
Invalid SAT score.!

Achievements:
Name:
Since:

Controller Hierarchy

Before you learn more about customizing Direct to Java Client applications, you should know what's going on behind the scenes.

In nondirect Java Client applications, user interfaces are stored in Interface Builder nib files. In Direct to Java Client applications, user interfaces are dynamically generated by the EOGeneration layer, which produces XML descriptions of the controllers in a user interface. Each user interface element in a Direct to Java Client application is managed by a controller. Multiple controllers are organized in a controller hierarchy, which defines the complete functionality of the application.

There is an application-wide controller hierarchy with an EOApplication object at its root. Each window or modal dialog in an application is defined by a more granular controller hierarchy. The controller hierarchies for windows or modal dialogs are referred to as the application's subcontrollers. Window controllers and modal dialog controllers have subcontrollers of their own such as text fields, table views, and checkboxes.

Controllers

The objects in the controller hierarchy are instances of EOController subclasses. The EOController class defines basic controller behavior. Collectively, controllers are responsible for managing the controller hierarchy (which includes building, connecting, and traversing the hierarchy) and handling actions. Controllers define and know how to respond to the actions users can perform.

Fundamentally, controllers are your best friend. They save you from writing raw Swing or from needing to maintain user interfaces in Java. They allow you to define user interfaces in XML descriptions with a common set of tags that are mapped into Swing-specific user interface elements and characteristics. Controllers also save you from needing to interact with Enterprise Objects programmatically. They automatically construct qualifiers and fetch specifications, fetch data into editing contexts, and in general take care of all the heavy lifting required in an application that uses Enterprise Objects.

The EOController subclasses fall into the following categories:

- **Application-level controllers** define application-level functionality. They define actions such as Quit and Save. Additionally they provide document management support such as tracking documents with unsaved changes. An application-level controller (such as `EOApplication` or `EODynamicApplication`) is the root of an application's controller hierarchy.
- **User interface–level controllers** manage portions of an application's user interface, such as windows (`EOWindowController`) and tab views (`EOTabViewController`). They determine the layout of their subcontrollers, resizing behavior, and so on.
- **Entity-level controllers** specify the user interface for performing a particular task on an entity. Entity-level controllers determine the functionality for querying, listing, and editing objects. They include `EOQueryController` and `EOListController`.
- **Property-level controllers** manage widgets for displaying properties. They provide widgets for entering text, displaying properties in a table, and so on. They include `EOTextFieldController` and `EOTableColumnController`.

Creating the Controller Hierarchy

The process for creating the controller hierarchy involves a `com.webobjects.eogeneration.EOControllerFactory` object, an application object, the rule system, and D2WComponent objects.

In a Direct to Java Client application, the client-side application object (`com.webobjects.eogeneration.EODynamicApplication` or a subclass of it) initializes the controller factory object. The controller factory, which lives on the client side, makes a request to the rule system on the server side for a particular controller or controller hierarchy. The rule system analyzes the application's data models (`.eomodeld` files) and rule models (`.d2wmodel` files) and generates an XML description of the controller or controller hierarchy that the controller factory requested.

The rule system then sends the XML descriptions it generated back to the controller factory. The controller factory then builds a user interface based on the XML descriptions it receives (it parses the XML using a `com.webobjects.eoapplication.EOXMLUnarchiver` object) .

The EOXMLUnarchiver maps XML tags to EOController classes, as illustrated in Table 6-2 (page 120).

Table 6-2 A subset of the controllers available in Direct to Java Client

XML tag	Controller class
`MODALDIALOGCONTROLLER`	EOModalDialogController
`ACTIONBUTTONSCONTROLLER`	EOActionButtonsController
`QUERYCONTROLLER`	EOQueryController
`TEXTFIELDCONTROLLER`	EOTextFieldController

XML tag	Controller class
`LISTCONTROLLER`	EOListController
`TABLECONTROLLER`	EOTableController
`TABLECOLUMNCONTROLLER`	EOTableColumnController

As an XML unarchiver creates the controller hierarchy, it configures the controllers according to the specified XML attribute values. For example, two of the XML attributes for EOTextField are `valueKey` and `isQueryWidget`:

```
<TEXTFIELDCONTROLLER valueKey="name" isQueryWidget="true"/>
```

These attributes correspond to the EOTextField methods `setValueKey` and `setIsQueryWidget`. The `valueKey="name"` attribute specifies that the text field controller corresponds to a property named “name.” The `isQueryWidget="true"` attribute specifies that the text field is used to get search criteria from the user and is not to display and edit a property’s value.

As users interact with the user interface, they may perform actions that require additional controllers. These requests are sent to the controller factory which communicates with the rule system on the server side, generates XML descriptions of controllers, and sends them back to the client.

For more information on the XML tags and attributes for controller classes, see “Controllers and Actions Reference” (page 265).

Using Rules in the Rule System

As well as understanding the role of controllers in Direct to Java Client applications, you need to know a bit more about the rule system. The default rule system in Direct to Java Client applications includes over one hundred rules. You can customize these rules and write new rules, too. So you need to know both how to leverage the default rules in your application and how to write custom rules.

Every Java Client class that can exist as part of an XML description for Direct to Java Client user interfaces includes XML identifiers. These identifiers come in the form of a single XML tag and one or more XML attributes.

For instance, EOComponentController’s XML tag is `COMPONENTCONTROLLER`, and its XML attributes include `alignmentWidth`, `iconName`, and `verticallyResizable`. This book includes a complete list of Java Client classes that have XML tags and XML attributes in “Controllers and Actions Reference” (page 265).

For example, when using a Direct to Java Client application, you may want to change the behavior of the query window. It’s not uncommon to want to query for all records in a particular entity, and the dialog asking if you want to search for all records can become repetitive. To see if the query window has any options for controlling its behavior, you’d first consult its XML attributes as found in “Controllers and Actions Reference” (page 265).

You’d find that the EOQueryController class includes an XML attribute called `runsConfirmDialogForEmptyQualifiers`. This attribute controls the confirmation dialog when you click Find in a query window without qualifying the search criteria. `runsConfirmDialogForEmptyQualifiers` is a Boolean attribute, so setting it to `false` disables the confirmation dialog.

You add this rule to your application's `d2w.d2wmodel` file using Rule Editor. You add the `d2w.d2wmodel` file to a project by making a new file of type "Empty File," naming it `d2w.d2wmodel` and associating it with the Application Server target.

Open your application's `d2w.d2wmodel` file and add a rule with these attributes:

Left-Hand Side:	`*true*`
Key:	`runsConfirmDialogForEmptyQualifiers`
Value:	`"false"`
Priority:	`50`

In this case, you don't need to specify the qualifier since only one controller has the `runsConfirmDialogForEmptyQualifiers` value. If you want to disable the confirmation dialog just for a specific entity, you can add this argument to the left-hand side: `entity.name="<entityName>"`.

Note: Be careful about using this rule without specifying an entity. The confirmation dialog is intended to avoid unqualified fetches on entities with a large number of records. Disabling the confirmation dialog for all entities in an application runs the risk of severely degrading both your database's performance and your application's usability as users are more likely to invoke unqualified searches.

See "Inside the Rule System" (page 163) for an explanation of rule priorities and for more general information on the rule system. Also see "Common Rules" (page 193) for examples of custom rules.

Additional Actions

Adding actions to Direct to Java Client applications is rather easy. There are four recommended procedures:

- Use Assistant to specify a new property with an EOActionController widget (for actions on enterprise objects; the action method is in a client-side business logic class).
- Subclass a controller class and write a rule to use it in the application (the action method is in the subclass).
- Write a custom controller class and include it in an XML description (for actions on user interface; the action method is in the custom controller class).
- Edit XML by hand to include an EOActionController with an `actionKey` tag specifying the action method (for actions on enterprise objects; the action method is in the client-side business logic).

Write the Action (Build a WOComponent)

Before you take steps to customize the application to invoke a new action, you need to write the code for the action. The action you'll add here sends the contents of a Student record to a specified email address. The code that constructs the email exists in your application's `Session.java` class. Rather than send a plain text email, the email sent is a WebObjects component email. This means that you can use a dynamic WOComponent object to populate the contents of the email.

Note: You'll better understand this part of the tutorial if you're familiar with the concepts involved in an HTML WebObjects application. The book *WebObjects Web Applications Programming Guide* is a great place to start learning.

Follow these steps to make the new WOComponent:

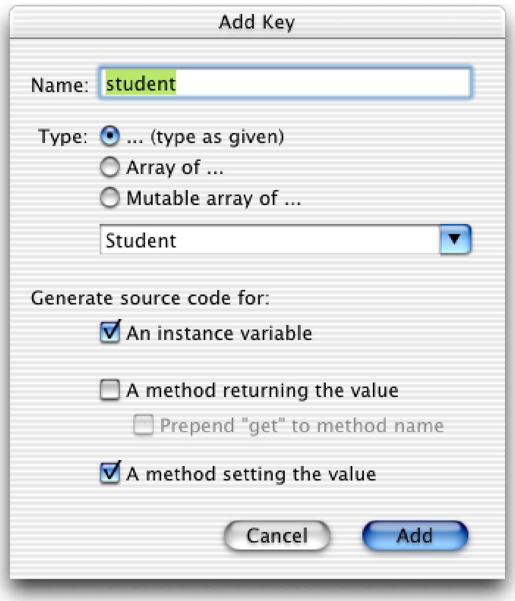
1. Make a new WOComponent in Project Builder. Choose New File from the File menu and select Component from the WebObjects list. Name the component `Report` and add it the Application Server target.

2. Open the component in WebObjects Builder and add a new key called `student` of type `Student`, as shown in Figure 6-13. Follow these steps:

   - Double-click `Report.wo` in Project Builder to open it in WebObjects Builder.
   - Choose Add Key from the Edit Source pop-up menu at the bottom of the window.

   Select the options to generate source code for an instance variable and a method setting the value.

Figure 6-13 New key of type Student in the Report component



Add Key

Name: student

Type: [✓ ... (type as given)] Array of ... Mutable array of ...

Student

Generate source code for:

[✓ An instance variable]

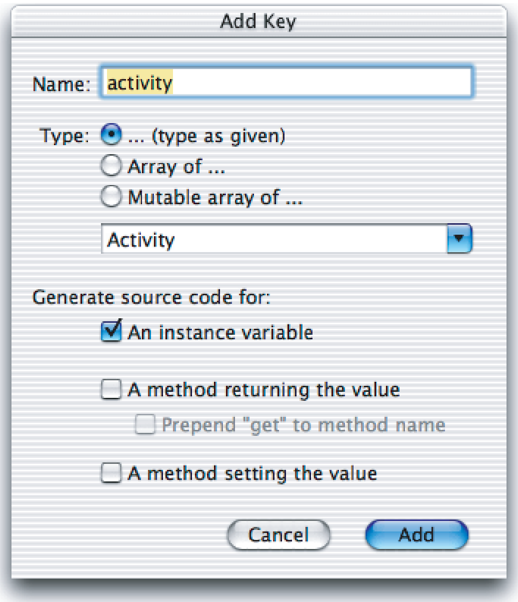
A method returning the value

Prepend "get" to method name

[✓ A method setting the value]

3. Add another new key called `activity` of type `Activity`, as shown in Figure 6-14. Select the option to generate source code for an instance variable.

Figure 6-14 New key of type Activity in the Report component



Name: activity

Type: [✓ ... (type as given)] Array of ... Mutable array of ...

Activity

Generate source code for:

[✓] An instance variable

[ ] A method returning the value

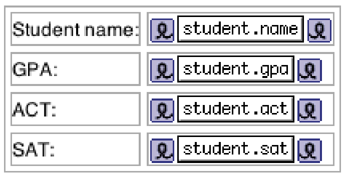
[ ] Prepend "get" to method name

[ ] A method setting the value

4. Add dynamic elements for Student's attributes. Add WOStrings for the `gpa`, `act`, `sat`, and `name` attributes as shown in Figure 6-15. They are shown here in a table, but that is optional.

   - Choose WOString from the WebObjects menu.
   - Drag from an attribute in the student key (such as `student.name`) to the center of the WOString element to bind the attribute to the element's `value` binding.
   - Repeat for all four elements.

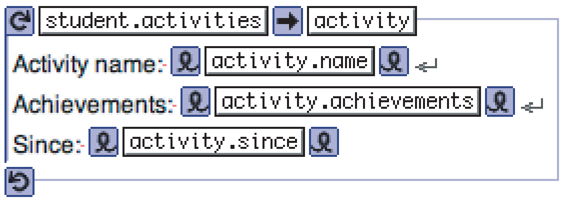
Figure 6-15 Dynamic elements for Student's attributes



Student name:	student.name
GPA:	student.gpa
ACT:	student.act
SAT:	student.sat

5. Add dynamic elements for Student's `activities` relationship. Add a WORepetition with `list = student.activities` and `item = activity`. Add WOStrings for `activity.name`, `activity.achievements`, and `activities.since` as shown in Figure 6-16.

Figure 6-16 WORepetition for Student's activities



6. Add this method to `Session.java` to compose and send the message:

```
public void clientSideRequestSendRecordViaEmail(EOEnterpriseObject record) {
        String messageSubject, messageBody, message;
        NSMutableArray recipients = new NSMutableArray();
        recipients.addObject("person@foo.com");

        Report report = new Report(context());
        report.setStudent(record);

        messageSubject = "Student report for " + record.valueForKey("name");
         message =
WOMailDelivery.sharedInstance().composeComponentEmail("sender@foo.com",
recipients, null, messageSubject, report, true);
}
```

This method uses the `com.webobjects.appserver.WOMailDelivery` class to send an email message containing information from a student record. You'll notice that the method is named `clientSideRequestSendRecordViaEmail` to conform to the default rules for remote method invocation.

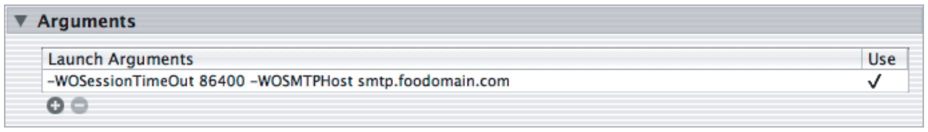
7. *This step is necessary only in WebObjects 5.0.x and 5.1.x.* Since the email is sent via remote method invocation, you need to provide a distribution layer delegate method in `Session.java` to allow the invocation. When the distribution layer starts up, it sets its delegate to the Session object, which allows you to override the methods defined in the EODistributionContext.Delegate class.

   In `Session.java`, add an import statement for the `com.webobjects.eodistribution` package and then add the distribution layer delegate method:

```
public boolean distributionContextShouldFollowKeyPath(EODistributionContext
        distributionContext, String path) {
        return (path.equals("session"));
}
```

8. You need to add a launch argument to the application representing the email server through which to send the message. Add `-WOSMTPHost` to your launch arguments with the name of a mail server on your network, as shown in Figure 6-17. Refer to "Add a Launch Argument" (page 67) if you've forgotten how to add a launch argument.

Figure 6-17 Add launch argument for SMTP host



```
-WOSessionTimeOut 86400 -WOSMTPHost smtp.foodomain.com
```

✓

You can now add custom actions to invoke the email composition. How the `clientSideRequestSendRecordViaEmail` method in `Session.java` is invoked depends on how you add the custom action. The following four sections describe the possibilities, in order of recommendation.

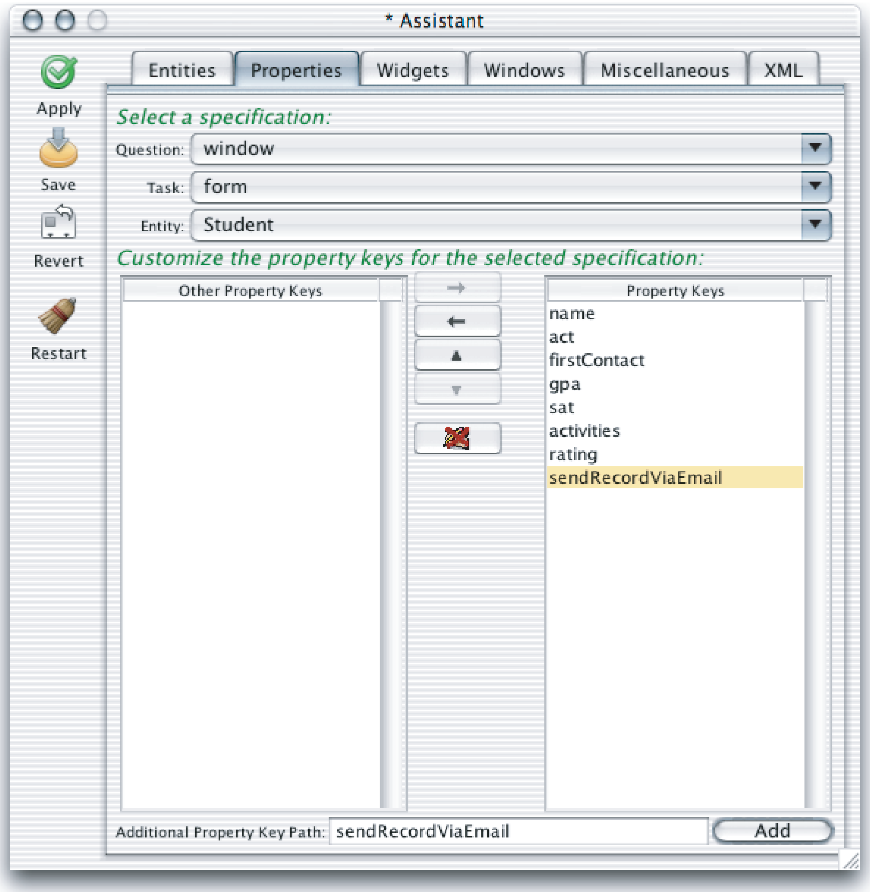
Use Assistant

Using Assistant is the easiest, fastest, but least flexible way to add an action to an application. Follow these steps:

1. Build and run the Admissions application and open Assistant.

2. Switch to the Properties pane and choose Question > window, Task > Form, and Entity > Student. Add a new property key called `sendRecordViaEmail` for Question=window, Task=form, Entity=Student. Do this using the Additional Property Key Path field. See Figure 6-18.

Figure 6-18 Add property key for new action



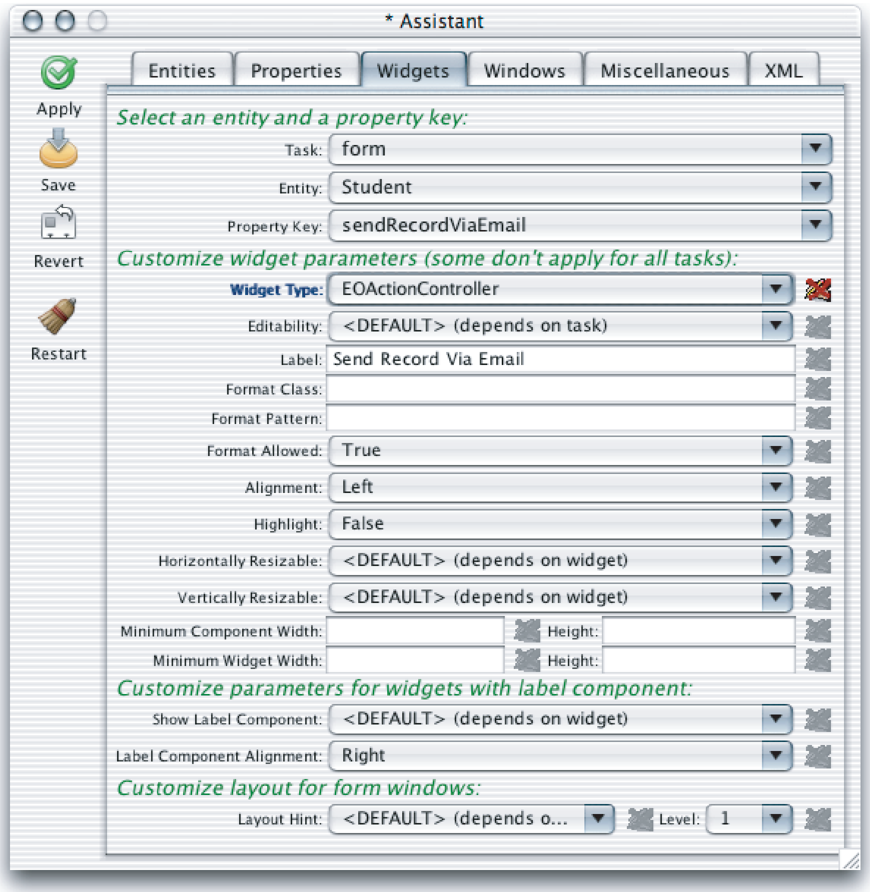
Question: window
Task: form
Entity: Student

Property Keys:
name
act
firstContact
gpa
sat
activities
rating
sendRecordViaEmail

Additional Property Key Path: sendRecordViaEmail

3. Switch to the Widgets pane, choose Task > form, Entity > Student, and Property key > sendRecordViaEmail. In the Widget Type pop-up menu, choose EOActionController as shown in Figure 6-19.

Figure 6-19 Change the widget type of the new property key



Select an entity and a property key:

Task: form
Entity: Student
Property Key: sendRecordViaEmail

Customize widget parameters (some don't apply for all tasks):

Widget Type: EOActionController
Editability: <DEFAULT> (depends on task)
Label: Send Record Via Email
Format Class:
Format Pattern:
Format Allowed: True
Alignment: Left
Highlight: False
Horizontally Resizable: <DEFAULT> (depends on widget)
Vertically Resizable: <DEFAULT> (depends on widget)
Minimum Component Width: Height:
Minimum Widget Width: Height:

Customize parameters for widgets with label component:

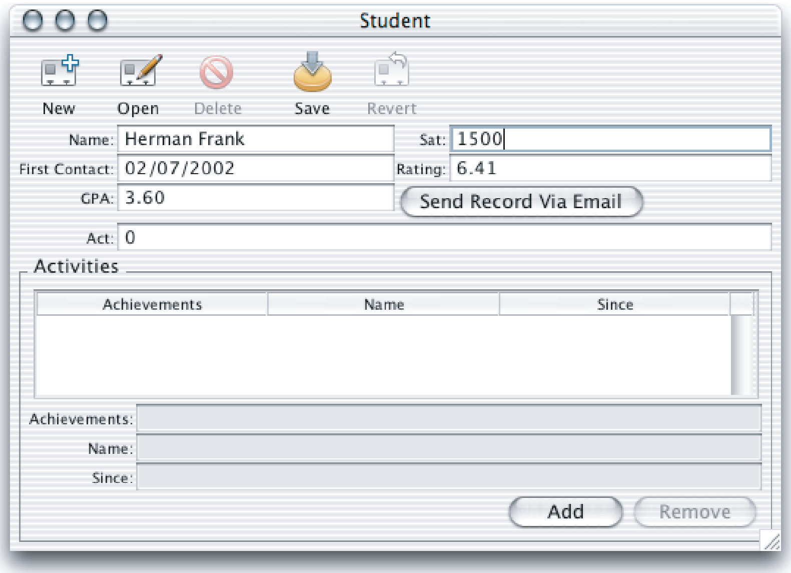
Show Label Component: <DEFAULT> (depends on widget)
Label Component Alignment: Right

Customize layout for form windows:

Layout Hint: <DEFAULT> (depends o... Level: 1

4. Save the changes and restart the client application from Assistant and you'll see a new button called Send Record Via Email in form windows for the Student entity as shown in Figure 6-20. Since it's an EOActionController defined in the Student entity, it invokes a method of the same name, `sendRecordViaEmail`, in the client-side business logic class for that entity (`businesslogic.client.Student` in this case).

Figure 6-20 The new property key as an EOActionController



Name: Herman Frank
Sat: 1500
First Contact: 02/07/2002
Rating: 6.41
GPA: 3.60
Act: 0

Achievements	Name	Since

Achievements:
Name:
Since:

Make a new student record or open an existing record and click the new button. If you started the client application from the command line, you see an `IllegalArugmentException` is thrown, stating that the method `sendRecordViaEmail` can't be found. (In Mac OS X, client applications started automatically by the `WOAutoOpenClientApplication` mechanism send exceptions to the console.) So, you need to add it to your client-side business logic class.

Add this method in the client-side `Student.java` file:

```
public void sendRecordViaEmail() {
        _distributedObjectStore().invokeRemoteMethodWithKeyPath(new
           EOEditingContext(), "session", "clientSideRequestSendRecordViaEmail", new
           Class[] {EOEnterpriseObject.class}, new Object[] {this}, true);
}
```

This method invokes the method you added to your `Session.java` class. It sends the enterprise object from which the action originated (the `this` parameter) and pushes the state of the client-side editing context to the server-side editing context (the `true` parameter). See the API reference documentation for `invokeRemoteMethodWithKeyPath` for detailed descriptions of each parameter.

In the code listing above, you'll notice that the remote method invocation is made on an object returned from the method `_distributedObjectStore()`. You need to add this method to the client-side `Student.java` class:

```
private EODistributedObjectStore _distributedObjectStore() {
        EOObjectStore objectStore = EOEditingContext.defaultParentObjectStore();
       if ((objectStore == null) || (!(objectStore instanceof EODistributedObjectStore)))
       {
         throw new IllegalStateException("Default parent object store needs to be an
              EODistributedObjectStore");
      }
        return (EODistributedObjectStore)objectStore;
}
```

Client-side remote methods that are not invoked on business logic classes (on subclasses of EOCustomObject) are invoked on the client's distributed object store. For instance, in an EOGenericRecord subclass, you can use the method `invokeRemoteMethod(String`*`methodName`*`, Class[]`*`argumentTypes`*`, Object[]`*`arguments`*`)`, which invokes a method named *`methodName`* in the server-side EOGenericRecord subclass of the same name.

But, if you want to invoke a remote method that is not in the server-side business logic class corresponding to the client-side business logic class from where the remote method invocation originates, you need to invoke the remote method on the client's distributed object store, as the example above shows.

See the WebObjects API reference documentation for the `com.webobjects.eodistribution.client` package for more information on the distributed object store and the different varieties of remote method invocations. Also see "The Distribution Layer" (page 93)for an introduction to the distribution layer and remote method invocation.

Next, you need to add the import statement for the client-side EODistribution layer to the `Student.java` class:

```
import com.webobjects.eodistribution.client.*;
```

Build and run the application, open a Student record, and click the Send Record Via Email button. If you added your email address to the recipients in the code you added to `Session.java`, you should see an email in your inbox with the information in the selected record.

Extend a Controller Class

Using Assistant to add an action may not provide you with the flexibility you need. Furthermore, the methods you added in the last section are not really appropriate in business logic classes. They are better suited to a dedicated controller class.

Extending a controller class and writing a rule to use it is the best way to provide custom actions in your application. It is much more flexible than just using Assistant and it's much better than the next two options, which both require freezing XML. Anytime you freeze XML, you lose a lot of the dynamism of the rule system. This means, for instance, that you are not as able to use the rule system to localize your application or provide access controls via rules. Also, subclassing controller classes doesn't incur the costs associated with writing completely custom controllers.

The dynamically generated user interfaces in Java Client rely on a core set of classes: EOFormController, EOQueryController, and EOListController. You can take real advantage of WebObjects' excellent object-oriented design to extend these classes to provide custom behavior.

Add a new file to your application called `CustomFormController.java`. Add it to the Web Server target. Copy and paste the code for it, shown in Listing 6-1 (page 131).

Listing 6-1 CustomFormController code

```
package admissions.client;

import java.io.*;
import javax.swing.*;
import java.awt.*;
import com.webobjects.foundation.*;
import com.webobjects.eocontrol.*;
import com.webobjects.eointerface.*;
import com.webobjects.eoapplication.*;
import com.webobjects.eogeneration.*;
import com.webobjects.eodistribution.client.*;

public class CustomFormController extends EOFormController {

    public CustomFormController(EOXMLUnarchiver unarchiver) {
        super(unarchiver);
    }

    protected NSArray defaultActions() {
        Icon icon = EOUserInterfaceParameters.localizedIcon("ActionIconOk");
        NSMutableArray actions = new NSMutableArray();

        actions.addObject(EOAction.actionForControllerHierarchy("sendRecordViaEmail",
            "Send Record Via Email", "Send Record Via Email", icon, null, null, 300,
50,
            false));
        return EOAction.mergedActions(actions, super.defaultActions());
    }

    public boolean canPerformActionNamed(String actionName) {
        return actionName.equals("sendRecordViaEmail") ||
                 super.canPerformActionNamed(actionName);
    }

    public void sendRecordViaEmail() {
       _distributedObjectStore().invokeRemoteMethodWithKeyPath(new EOEditingContext(),
         "session","clientSideRequestSendRecordViaEmail", new Class[]
         {EOEnterpriseObject.class}, new Object[] { selectedObject()}, true);
    }


    private EODistributedObjectStore _distributedObjectStore() {
        EOObjectStore objectStore = EOEditingContext.defaultParentObjectStore();
       if ((objectStore == null) || (!(objectStore instanceof EODistributedObjectStore)))
         {
            throw new IllegalStateException("Default parent object store needs to be an
              EODistributedObjectStore");
        }
        return (EODistributedObjectStore)objectStore;
    }

}
```

When you examine this code, you'll notice that two of its methods are those you added in the last section. So you can remove both `sendRecordViaEmail` and `_distributedObjectStore` from the client-side `Student.java` class. The `defaultActions` method adds to the application's actions and `canPerformActionNamed` authorizes the invocation of the `sendRecordViaEmail` method.

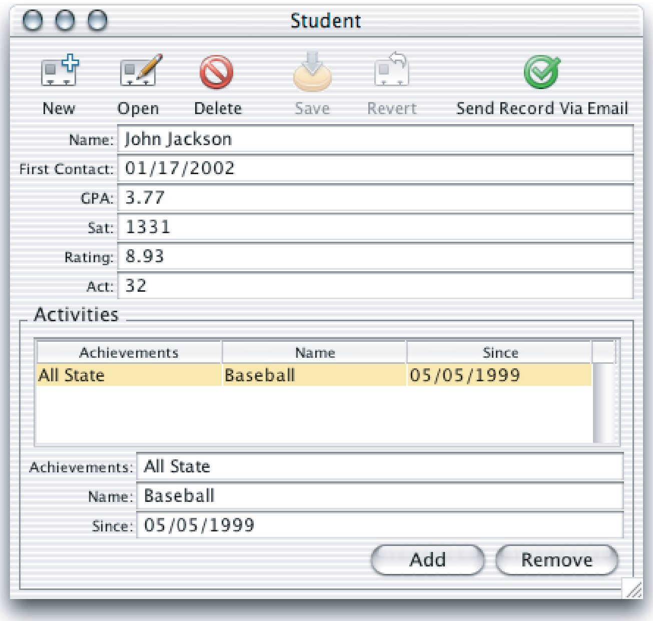
To use this class in form windows for the Student entity, you need to add a rule to the project's `d2w.d2wmodel` file:

Left-Hand Side:	`((task='form') and (controllerType='entityController') and (entity.name='Student'))`
Key:	`className`
Value:	`"admissions.client.CustomFormController"`
Priority:	`50`

You add the `d2w.d2wmodel` file to a project by making a new file of type "Empty File," naming it `d2w.d2wmodel`, and associating it with the Application Server target. Control-click the file after adding it to the project to display its contextual menu. Choose "Open with Finder" to open the file in Rule Editor. Then add the rule shown above.

Build and run the application and remove the action you added with Assistant (you can either go to the Direct to Java Client Assistant and move the action to the Other Property Keys list or find the rule in the `user.d2wmodel` file and delete it by hand). If successful, form windows for the Student entity should look like Figure 6-21.

Figure 6-21 Image form window with new buttons



Name: John Jackson
First Contact: 01/17/2002
GPA: 3.77
Sat: 1331
Rating: 8.93
Act: 32

Achievements	Name	Since
All State	Baseball	05/05/1999

Achievements: All State
Name: Baseball
Since: 05/05/1999

Clicking the Send Record Via Email button should send an email with the current record's information to the recipients you declared in the method in `Session.java`, which constructs and sends the email.

Additional Exercise

For the custom action that sends a record via email, you may find that hard-coding the email recipients is not ideal. Rather, you might want the flexibility of choosing the recipients on a per-record basis. By using the controller factory programmatically, this is actually quite simple.

First, following the Model-View-Controller paradigm, you need to write a new class to display a dialog in which the user can select the email recipients. Although you could save writing a few lines of code by putting the controller factory invocation in the business logic class, this is bad design. Business logic classes (enterprise objects) should not include any user interface code. So, add a new client-side class called SelectEmail to your project :

```
package admissions.client;

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

public class SelectEmail extends Object{

    public SelectEmail() {
        super();
    }

    public NSArray selectEmailAddresses() {
        return
EOControllerFactory.sharedControllerFactory().selectWithEntityName
("Email", true, false);
    }
}
```

The class is rather simple and contains a single method that invokes a method on the controller factory. This displays a selection dialog for the Email entity as shown in Figure 6-22 (page 135).

The second argument in the `selectWithEntityName` method (`true`) allows multiple selection in the select dialog so you can choose multiple email addresses. The method returns the objects that are selected in the selection dialog.

Before you see any email addresses in that dialog, however, you have to add an entity to your EOModel called "Email", generate SQL for it, and add entries to it. Create a new entity and add to it two attributes:

- `emailID`; column name `EMAIL_ID`; external type `int`; internal type `Integer`; mark as the primary key; don't mark as either kind of class property
- `address`; column name `ADDRESS`; external type `char`; internal type `String`; mark as both a client-side and server-side class property

When you generate SQL for the Email entity, select only Create Tables and Primary Key Constraints.

The Email entity is considered an Enumeration entity by the rule system, so you can add data to it by choosing Enumeration Window from the Tools menu in the client application.

Next, you need to modify the `sendRecordViaEmail` action method in `CustomFormController.java` as shown:

```
public void sendRecordViaEmail() {
        SelectEmail select = new SelectEmail();
        NSArray globalIDs = select.selectEmailAddresses();

        _distributedObjectStore().invokeRemoteMethodWithKeyPath(new
          EOEditingContext(),"session", "clientSideRequestSendRecordViaEmail", new
          Class[] {EOEnterpriseObject.class, NSArray.class}, new Object[]
          {selectedObject(), globalIDs}, true);
 }
```

These modifications to `CustomFormController.java` instantiate a new SelectEmail object and invoke the method to display the dialog that allows users to select the email addresses to send the current report to.

The remote method invocation now sends the selected email address (represented by the `globalIDs` object) and the report from which the `sendRecordViaEmail` action was invoked (represented by the objects returned from the `selectedObject()` method in the remote method invocation) to the method `clientSideRequestSendRecordViaEmail` in the `Session.java` class on the server.

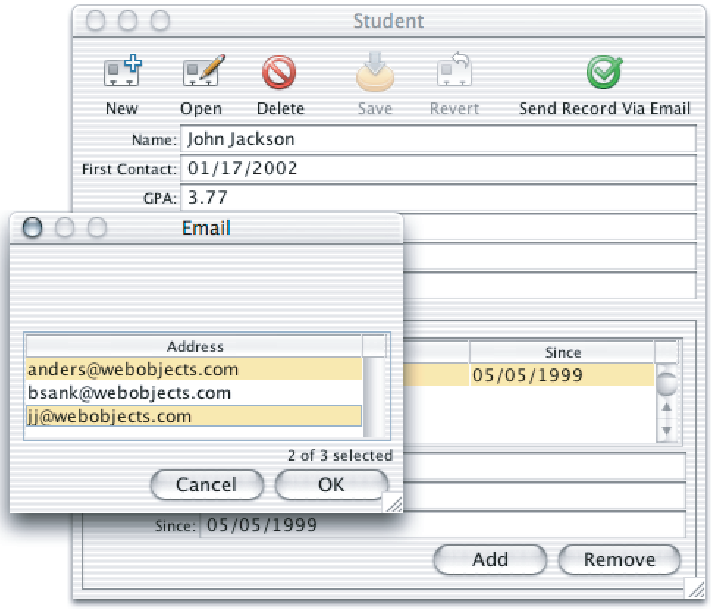
Next, you need to modify the `clientSideRequestSendRecordViaEmail` method in the server-side `Session.java` class to accept the new `globalIDs` argument:

```
public void clientSideRequestSendRecordViaEmail(EOEnterpriseObject record, NSArray
                                                                   sendTo) {
        String messageSubject, messageBody, message;
        NSMutableArray recipients = new NSMutableArray();
        //recipients.addObject("person@foo.com");

        java.util.Enumeration e = sendTo.objectEnumerator();
        while (e.hasMoreElements()) {
            EOEnterpriseObject email =
defaultEditingContext().objectForGlobalID((EOGlobalID)e.nextElement());
            String emailAddress = (String)email.valueForKey("email");
            recipients.addObject(emailAddress);
        }

        Report report = new Report(context());
        report.setStudent(record);

        messageSubject = "Student report for " + record.valueForKey("name");
        message =
WOMailDelivery.sharedInstance().composeComponentEmail("sender@foo.com",
recipients, null, messageSubject, report, true);
}
```

Instead of statically setting the array recipients, the array is set dynamically to the email addresses passed in by the `sendTo` array.

Build and run the application. Open a student record and click Send Record Via Email. A dialog like that shown in Figure 6-22 should appear. Select some email addresses and click OK. Check your email to see if you are successful.

Figure 6-22 Choose email recipients



Name: John Jackson
First Contact: 01/17/2002
GPA: 3.77

Address
anders@webobjects.com
bsank@webobjects.com
jj@webobjects.com

2 of 3 selected

Since
05/05/1999

Since: 05/05/1999

Debugging

As you use more difficult customization techniques, you'll need more debugging information. Direct to Java Client applications consist of much more than Java code. So, you need tools to help you debug the other main aspects: database access and the rule system.

You can see the SQL messages passed to the database by adding `-EOAdaptorDebugEnabled YES` to your launch arguments on the server application. By adding `-D2WTraceRuleFiringEnabled YES` to your launch arguments, you can see all the rule system rules and your custom rules as they are fired.

If those two flags don't provide you with enough information, you can add `-NSDebugGroups -1` and `-NSDebugLevel 3`, which activate logging for the internal workings of WebObjects. Using `-NSDebugGroups -1` gives you debug logging information for all aspects of the system. By specifying specific debug groups, you can narrow down the amount of information logged. See the API reference for NSLog for more information on how to use NSLog.

Nondirect Java Client Development

The direct approach to building Java Client applications and its customization techniques should allow you to sufficiently customize your application's user interface. However, it is possible and often useful to build completely customized Java Client user interfaces in Interface Builder. This chapter teaches you how to build custom user interfaces.

Building Custom Interfaces

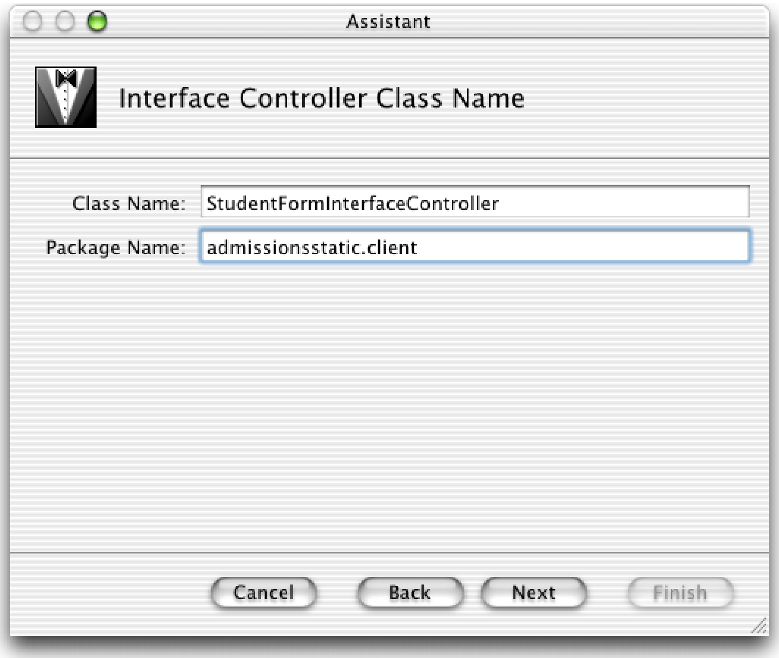
You create nondirect Java Client applications in Project Builder using the Java Client Application project type.

Make a new Java Client project called `AdmissionsStatic`. Add the EOModel file from the last tutorial.

In the Interface Controller Class Name pane, the interface controller class name should be StudentFormInterfaceController as shown in Figure 7-1. Make sure the package name is `admissionsstatic.client`. When creating Java Cilent interfaces, you must always specify the correct package name.

Note: Mac OS X 10.2 introduced a new nib file format. The nib file created by the Java Client setup assistant uses the pre-10.2 nib file format. The first time you try to save a nib file using the pre-10.2 file format, you'll be asked which format you want to use. When developing Java Client nibs, use the pre-10.2 file format.

Figure 7-1 Name the interface controller



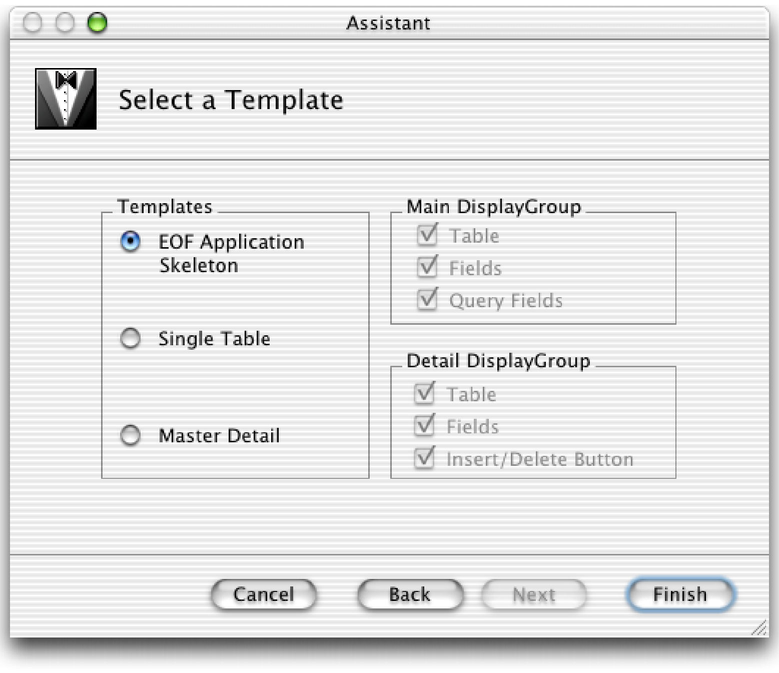
Class Name: StudentFormInterfaceController
Package Name: admissionsstatic.client

Add the `Admissions.eomodeld` file when prompted.

Select the fourth option in the Choose Download Classes pane (Download main bundle and custom framework classes). Optionally edit the fields in the Web Start pane.

In the Select a Template pane, select EOF Application Skeleton as shown in Figure 7-2.

Figure 7-2 Choose a template for the interface controller



Templates
- [✓ EOF Application Skeleton]
- Single Table
- Master Detail

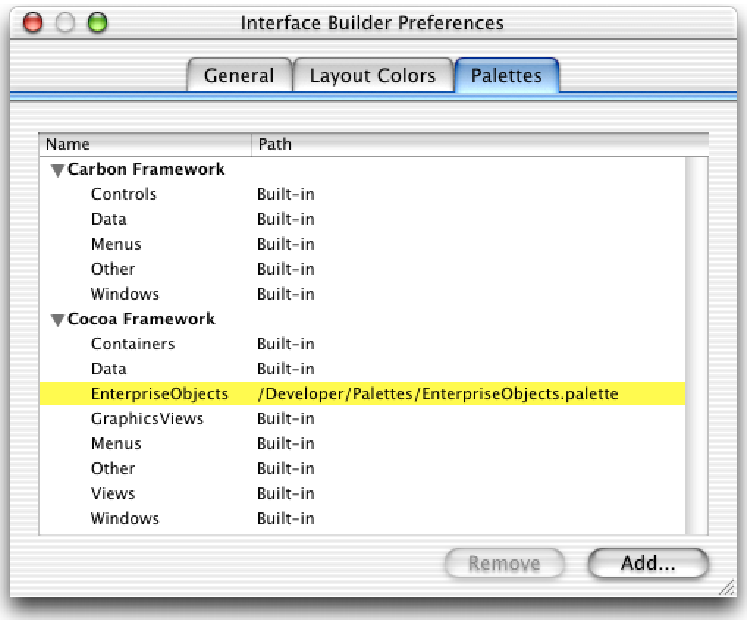
Main DisplayGroup
- ✓ Table
- ✓ Fields
- ✓ Query Fields

Detail DisplayGroup
- ✓ Table
- ✓ Fields
- ✓ Insert/Delete Button

For Java Client applications, Project Builder creates an Interface Builder file (`.nib`) and its associated Java class. By default, it's put in the Interfaces group. Double-click `StudentFormInterfaceController.nib` to open the file in Interface Builder.

Interface Builder needs a special palette to work with Java Client user interfaces. The EnterpriseObjects palette should load by default and appear in the Palettes pane of the Interface Builder Preferences window as shown in Figure 7-3.

Figure 7-3 Interface Builder palettes


Interface Builder Preferences

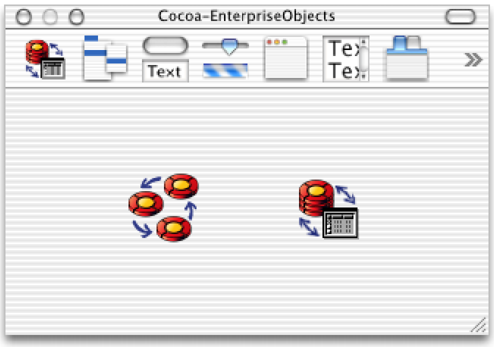
General | Layout Colors | Palettes

Name	Path
▼ Carbon Framework	
Controls	Built-in
Data	Built-in
Menus	Built-in
Other	Built-in
Windows	Built-in
▼ Cocoa Framework	
Containers	Built-in
Data	Built-in
EnterpriseObjects	/Developer/Palettes/EnterpriseObjects.palette
GraphicsViews	Built-in
Menus	Built-in
Other	Built-in
Views	Built-in
Windows	Built-in

Remove | Add...

If it does not appear, click the Add button, navigate to `/Developer/Palettes` and double-click `EnterpriseObjects.palette`. The palette should then appear in Interface Builder's palettes window as shown in Figure 7-4 (page 140).

Figure 7-4 Enterprise Objects palette



Laying Out the User Interface

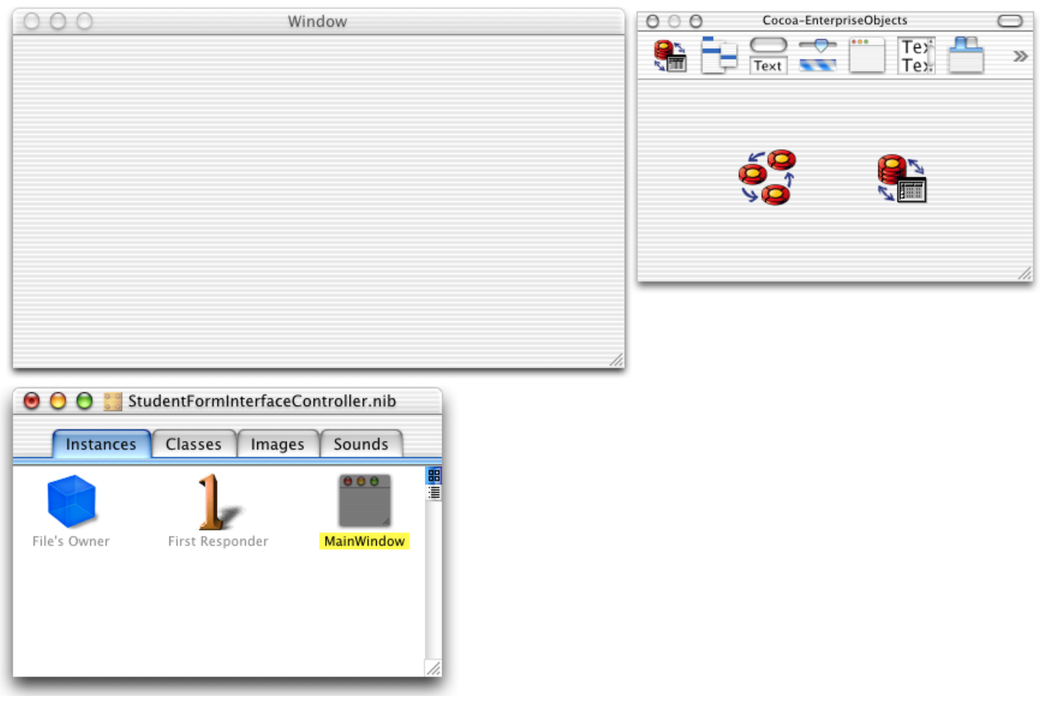
To create custom interfaces, you use Interface Builder, the same application used to build Cocoa desktop applications in Mac OS X. This tool gives you a wide variety of widgets to choose from, and most importantly, allows you to connect the user interface to objects in your data model.

The associations and connections you can make in Interface Builder make it the best tool for developing completely custom user interfaces for Java Client applications. You can write completely custom Java Client user interfaces in raw Swing or by using other third-party tools, but then you'll have to make all the associations and connections programmatically.

Interface Builder's integration with EOModeler allows you to easily build a user interface that is tightly coupled to your data model. It's as easy as dragging model elements from EOModeler into the content window in Interface Builder.

A blank window (which corresponds to the MainWindow object), a nib file window, and a palette window appear when Interface Builder launches, as shown in Figure 7-5.

Figure 7-5 The Interface Builder environment



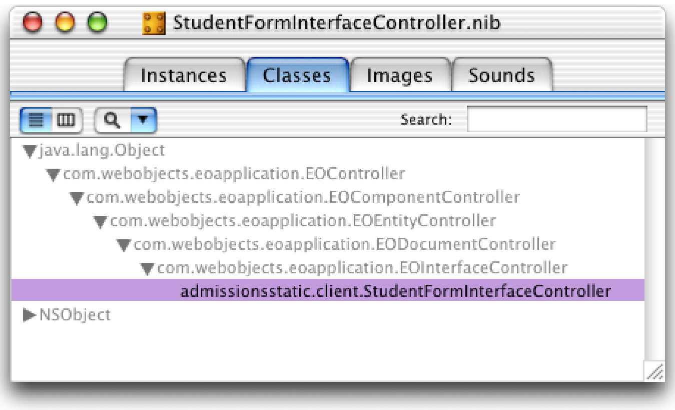
Prepare the Nib File

Note: This section is not necessary in WebObjects 5.2 as the nib file's controller classes are set correctly by the setup assistant in Project Builder when you add a nib to a project or create a nondirect Java Client project.

Before adding to the nib file, you may need to associate it with its controller class. This should happen automatically but if it doesn't, you must make the association manually.

While the nib file is open in Interface Builder, click the Classes tab of the nib file window. View the classes in inheritance mode (the vertical list), and click the disclosure triangle next to `java.lang.Object` to reveal the Java Client classes. Continue clicking disclosure triangles up through `com.webobjects.eoapplication.EOInterfaceController` as shown in Figure 7-6.

Figure 7-6 Classes pane in the nib file window

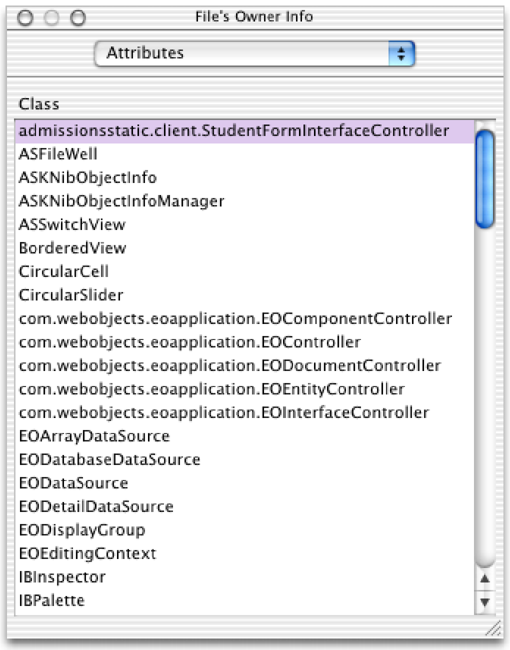


```
▼java.lang.Object
  ▼com.webobjects.eoapplication.EOController
    ▼com.webobjects.eoapplication.EOComponentController
      ▼com.webobjects.eoapplication.EOEntityController
        ▼com.webobjects.eoapplication.EODocumentController
          ▼com.webobjects.eoapplication.EOInterfaceController
              admissionsstatic.client.StudentFormInterfaceController
▶NSObject
```

Click `com.webobjects.eoapplication.EOInterfaceController` in the classes list and press Return. This subclasses EOInterfaceController and thus the new class inherits its targets and outlets. The name of the new subclass is the fully qualified name of the nib file, `admissionsstatic.client.StudentFormController`, as shown in Figure 7-6.

Now that you've created a new class, you must associate the nib file with it. To do this, switch to the Instances pane of the nib file window and click File's Owner. Choose Show Info from the Tools menu and choose Attributes from the pop-up menu. In the list of classes, select `admissionsstatic.client.StudentFormController` as shown in Figure 7-7.

Figure 7-7 Assign the custom subclass to File's Owner

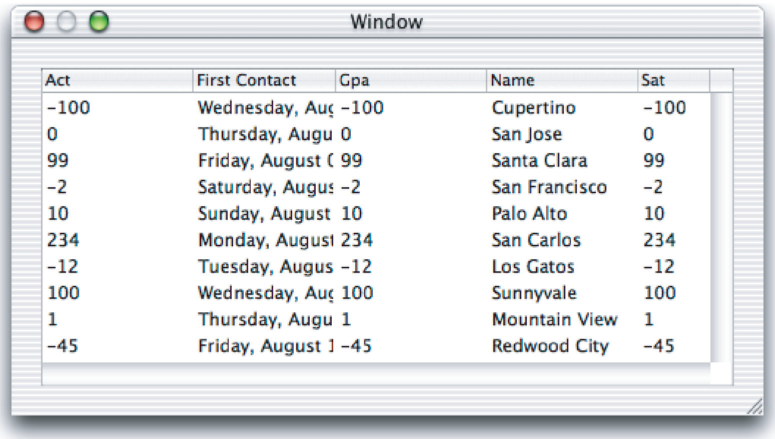


```
admissionsstatic.client.StudentFormInterfaceController
ASFileWell
ASKNibObjectInfo
ASKNibObjectInfoManager
ASSwitchView
BorderedView
CircularCell
CircularSlider
com.webobjects.eoapplication.EOComponentController
com.webobjects.eoapplication.EOController
com.webobjects.eoapplication.EODocumentController
com.webobjects.eoapplication.EOEntityController
com.webobjects.eoapplication.EOInterfaceController
EOArrayDataSource
EODatabaseDataSource
EODataSource
EODetailDataSource
EODisplayGroup
EOEditingContext
IBInspector
IBPalette
```

Integrate the Model

Open the `Admissions.eomodeld` from within the Admissions project to launch EOModeler. Then drag the Student entity from EOModeler into the main window in Interface Builder. The main window should then appear as in Figure 7-8.

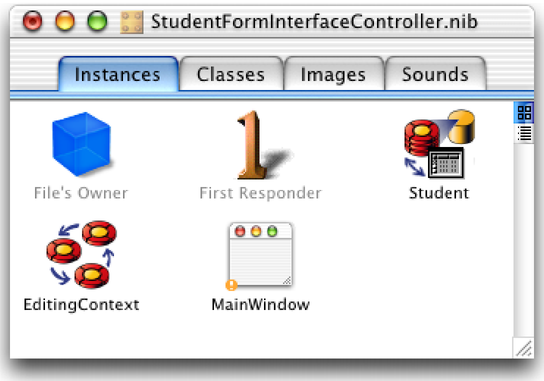
Figure 7-8 The Student entity dragged into Interface Builder



Act	First Contact	Gpa	Name	Sat
-100	Wednesday, Au	-100	Cupertino	-100
0	Thursday, Augu	0	San Jose	0
99	Friday, August	99	Santa Clara	99
-2	Saturday, Augus	-2	San Francisco	-2
10	Sunday, August	10	Palo Alto	10
234	Monday, August	234	San Carlos	234
-12	Tuesday, Augus	-12	Los Gatos	-12
100	Wednesday, Au	100	Sunnyvale	100
1	Thursday, Augu	1	Mountain View	1
-45	Friday, August 1	-45	Redwood City	-45

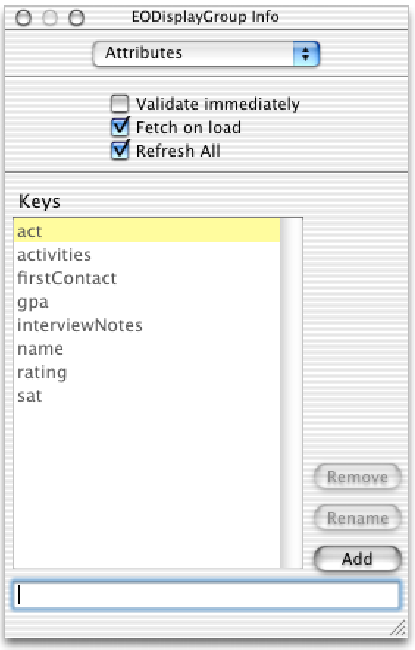
In the nib file window, there's now an EODisplayGroup object named "Student." The first display group you add to the model also adds an EOEditingContext object named "EditingContext" to the nib file window. The nib file window should appear as in Figure 7-9 (page 143).

Figure 7-9 Display group and editing context



You can set options for the Student display group by selecting it in the nib file window and choosing Show Info from the Tools menu. In the Attributes pane, make sure "Fetch on load" is selected as shown in Figure 7-10 (page 144). This option is important because it allows data to be fetched from the database when the application starts up.

Figure 7-10 Display group options in Interface Builder



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

Keys

act
activities
firstContact
gpa
interviewNotes
name
rating
sat

The keys listed in the EODisplayGroup Info window correspond to the class properties specified for the entity in EOModeler. You can add other keys that are not class properties such as methods you define in the associated Enterprise Objects class, as is done in "Using Pop-up Menus in Nib Files" (page 227).

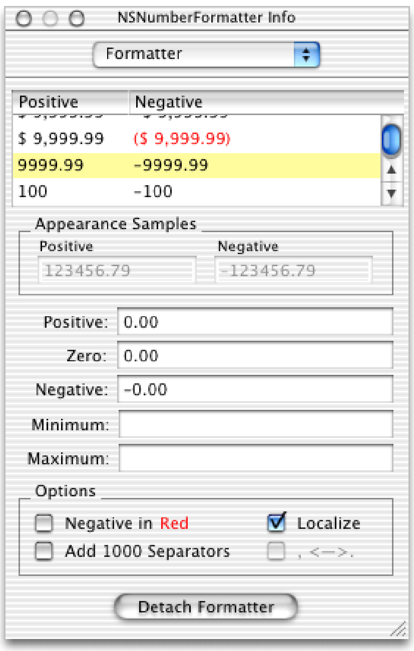
By dragging an entity from EOModeler into Interface Builder, you created a functional yet simple application. However, you should make some simple changes to improve it.

Add Formatters

The columns for `gpa` and `firstContact` are numeric, and you can set the numeric format style for column data directly in Interface Builder. If you don't, the `gpa` column defaults to an integer format, so the values will be rounded—making that data less relevant. The `firstContact` column defaults to a date format that includes the day of the week, information that is not particularly useful for that attribute in this application.

To change the formatters, double-click the `Gpa` column to bring up the Info window. Choose Formatter from the pop-up menu and select a formatter with a decimal point for the `Gpa` column as shown in Figure 7-11 (page 145).

Figure 7-11 Choose a formatter for the `Gpa` column



Positive	Negative
$ 9,999.99	($ 9,999.99)
9999.99	-9999.99
100	-100

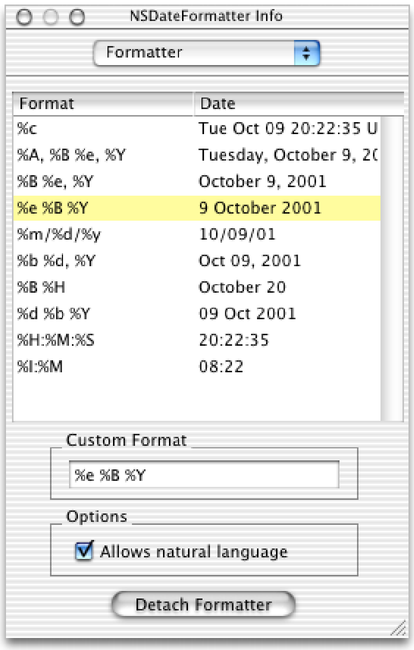
Appearance Samples: Positive 123456.79, Negative -123456.79

Positive: 0.00
Zero: 0.00
Negative: -0.00
Minimum:
Maximum:

Options: ☐ Negative in Red ☑ Localize ☐ Add 1000 Separators ☐ , <—>.

For the `FirstContact` column, select a simple date format as shown in Figure 7-12.

Figure 7-12 Choose a formatter for the `FirstContact` column



Format	Date
%c	Tue Oct 09 20:22:35 U
%A, %B %e, %Y	Tuesday, October 9, 2(
%B %e, %Y	October 9, 2001
%e %B %Y	9 October 2001
%m/%d/%y	10/09/01
%b %d, %Y	Oct 09, 2001
%B %H	October 20
%d %b %Y	09 Oct 2001
%H:%M:%S	20:22:35
%I:%M	08:22

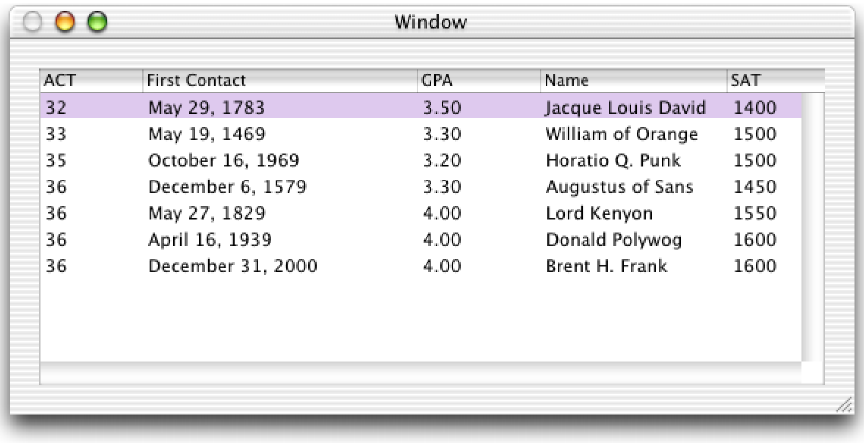
Custom Format: %e %B %Y

Options: ☑ Allows natural language

Finally, capitalize the column names so that they're as shown in Figure 7-13.

Interface Builder provides the ability to test the application. It actually connects to the database and fetches data. You can test it by choosing File > Test Interface.

Figure 7-13 Testing the application



ACT	First Contact	GPA	Name	SAT
32	May 29, 1783	3.50	Jacque Louis David	1400
33	May 19, 1469	3.30	William of Orange	1500
35	October 16, 1969	3.20	Horatio Q. Punk	1500
36	December 6, 1579	3.30	Augustus of Sans	1450
36	May 27, 1829	4.00	Lord Kenyon	1550
36	April 16, 1939	4.00	Donald Polywog	1600
36	December 31, 2000	4.00	Brent H. Frank	1600

Note that because “Fetch on load” is enabled for the Student EODisplayGroup, the data is automatically fetched when you test the interface.

Adding Action Methods

You can add basic behavior to your application, such as adding, deleting, and saving 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 a user clicks a button or another control object.

Add a button to the interface by dragging a button from the Views palette. Make three buttons named “Add,” “Remove,” and “Save.” These buttons will be used to insert new Student records, delete Student records, and save changes, respectively.

Connect the Add button to the EODisplayGroup’s `insert` method by Control-dragging from the Add button to the Student EODisplayGroup. Choose Outlets in the pop-up menu in the NSButton Info window. Select target in the left column and double-click the `insert:` outlet in the right column. See Figure 7-14 and Figure 7-15.

Figure 7-14 Connect the Add button to the `insert` method of the Student EODisplayGroup

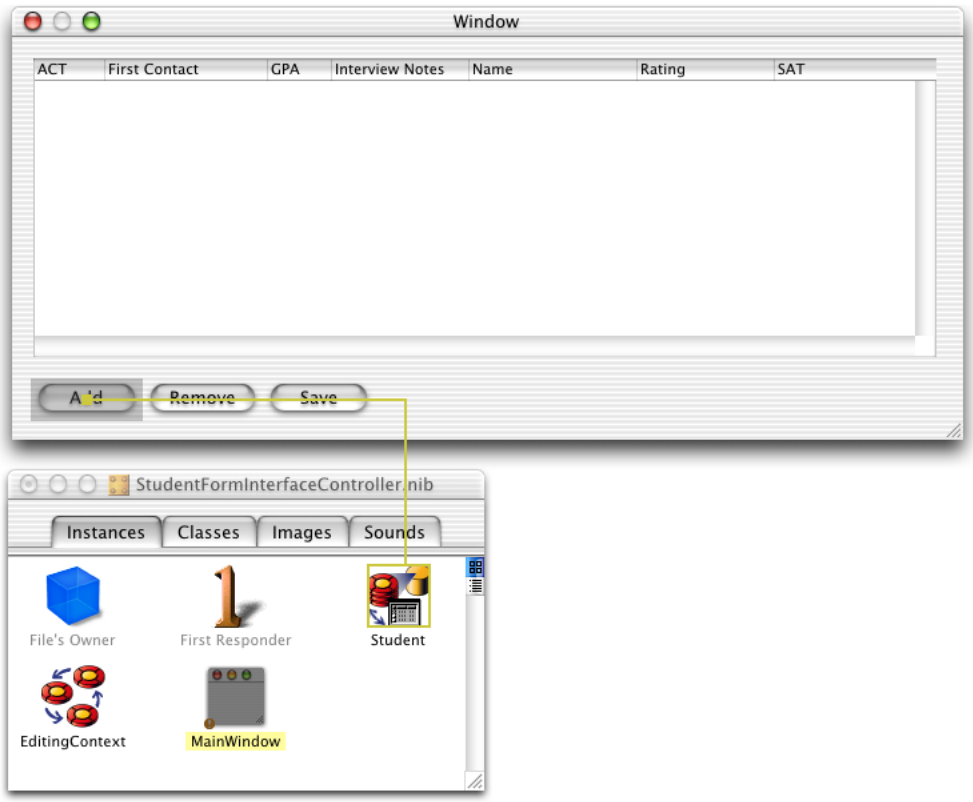
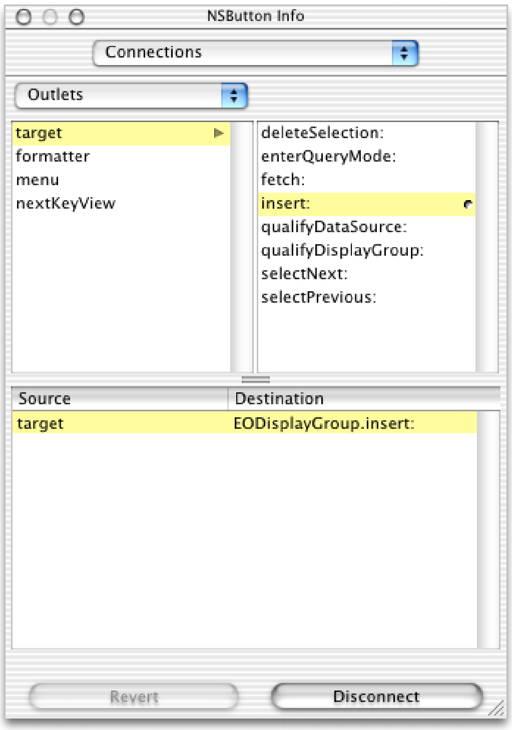


Figure 7-15 Select the `insert` method



Outlets	Actions
target ▶	deleteSelection:
formatter	enterQueryMode:
menu	fetch:
nextKeyView	insert:
	qualifyDataSource:
	qualifyDisplayGroup:
	selectNext:
	selectPrevious:

Source	Destination
target	EODisplayGroup.insert:

Using the same process, connect the Remove button to the `deleteSelection:` method. Finally, connect the Save button to the `saveChanges:` method in the EditingContext object.

Save the nib file. Build and run the project. You have a fully functional application with the capability to add, remove, and save records to the database.

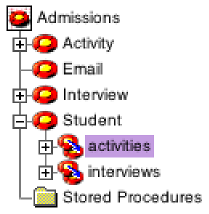
Create a Master-Detail Interface

To express the relationships in your EOModel, you use a master-detail interface. This interface includes a master table that holds records for the source of the relationship and a detail table that 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 table.

Before adding a master-detail interface, delete the table view from the nib file's main window and delete the Student EODisplayGroup and the EOEditingContext object from the nib file window.

You create a master-detail interface by simply dragging a relationship from EOModeler into a nib file window. Drag the Student entity's `activities` relationship from EOModeler onto the main window in Interface Builder. This creates a master-detail relationship. The icon you drag is found under the Student entity in the entity list pane of EOModeler. You may have to click the plus icon to show the relationship.

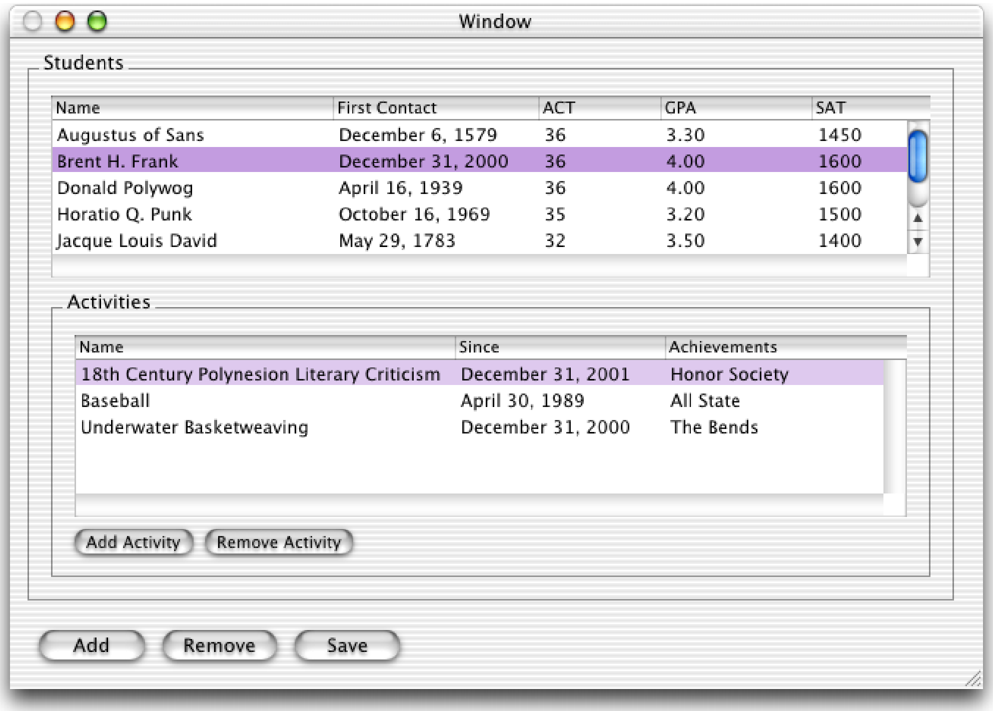
Figure 7-16 The `activities` relationship in the Student entity



Reconnect the Add and Remove buttons to the Student EODisplayGroup. Add two buttons for the detail part of the relationship to add and remove activities. Connect them to the activities display group, connecting to the `insert` and `deleteSelection` methods respectively. Add formatters to the columns as you did earlier.

Test the master-detail interface by choosing Test Interface from the File menu. Figure 7-17 (page 149) shows the master-detail interface with each table in a subview of a box view.

Figure 7-17 A master-detail interface



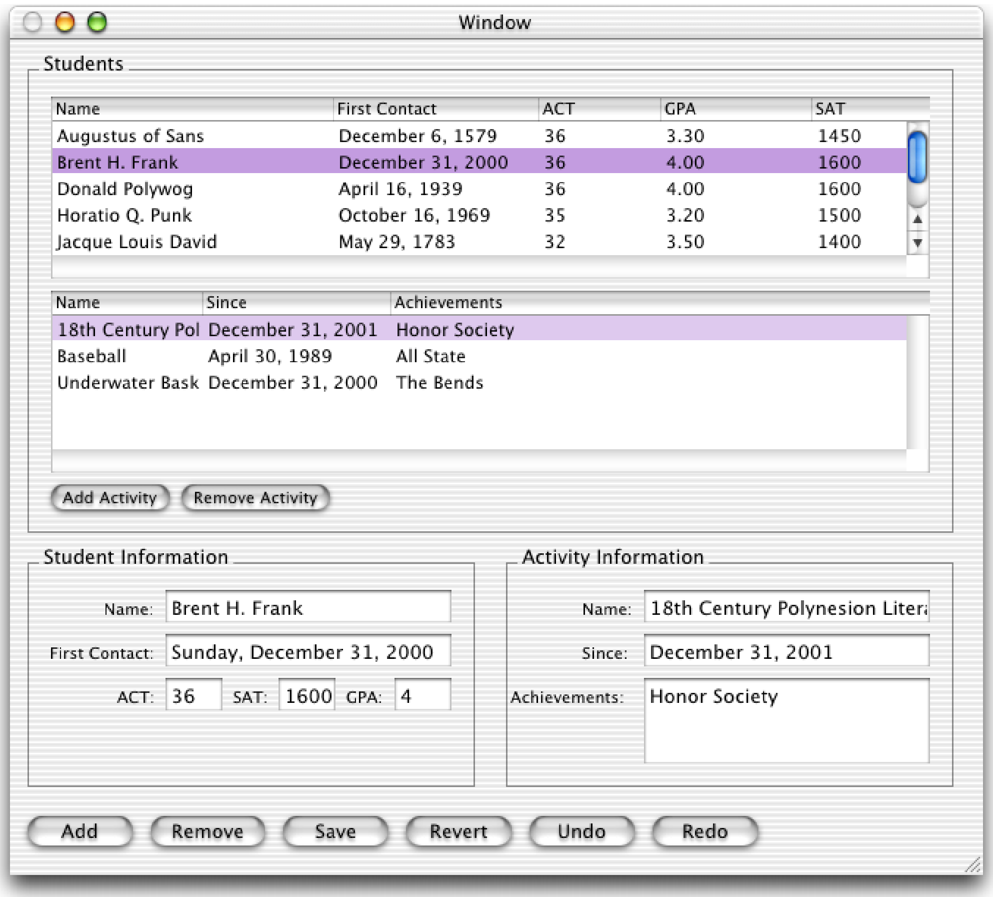
Name	First Contact	ACT	GPA	SAT
Augustus of Sans	December 6, 1579	36	3.30	1450
Brent H. Frank	December 31, 2000	36	4.00	1600
Donald Polywog	April 16, 1939	36	4.00	1600
Horatio Q. Punk	October 16, 1969	35	3.20	1500
Jacque Louis David	May 29, 1783	32	3.50	1400

Name	Since	Achievements
18th Century Polynesion Literary Criticism	December 31, 2001	Honor Society
Baseball	April 30, 1989	All State
Underwater Basketweaving	December 31, 2000	The Bends

The master-detail interface you just created can be improved. Although you can add new records by entering text directly in the table columns, it would be nice to provide text fields for doing the same thing. Also, you should take advantage of more built-in features of the technology, such as reversion, undo, and redo.

It's easy to add widgets in Interface Builder. Simply drag widgets from the Interface Builder palette onto the window. Figure 7-18 (page 150) illustrates the complete widget set of text fields, text areas, and buttons for the master-detail interface.

Figure 7-18 Complete widget set for the master-detail interface



Name	First Contact	ACT	GPA	SAT
Augustus of Sans	December 6, 1579	36	3.30	1450
Brent H. Frank	December 31, 2000	36	4.00	1600
Donald Polywog	April 16, 1939	36	4.00	1600
Horatio Q. Punk	October 16, 1969	35	3.20	1500
Jacque Louis David	May 29, 1783	32	3.50	1400

Name	Since	Achievements
18th Century Pol	December 31, 2001	Honor Society
Baseball	April 30, 1989	All State
Underwater Bask	December 31, 2000	The Bends

Student Information

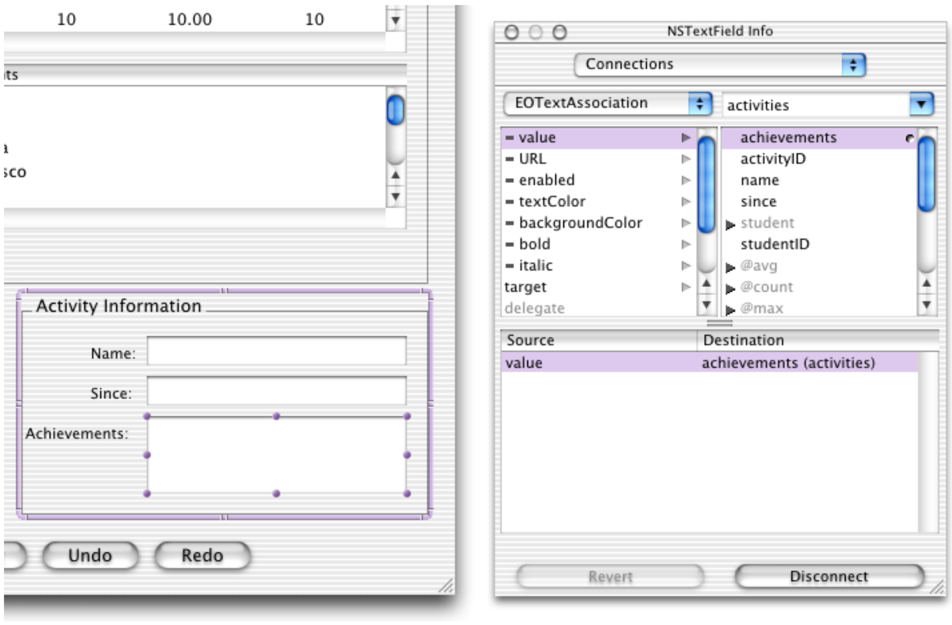
Name: Brent H. Frank
First Contact: Sunday, December 31, 2000
ACT: 36 SAT: 1600 GPA: 4

Activity Information

Name: 18th Century Polynesion Liter
Since: December 31, 2001
Achievements: Honor Society

Once you drag a widget into a window, you must connect it to the application. Consider the Achievements text field for the Activities entity. Once it's placed in the interface, Control-drag from the text field to the Activities entity in nib file window. In the Info window, choose EOTextAssociation from the pop-up menu and double-click "achievements" in the scrolling list, as shown in Figure 7-19 (page 151). This creates an association between the widget and the attribute in the entity. So, when you select a record, the value of the `achievements` attribute for that record is also displayed in the text field. This also allows you to edit the value of the attribute with which a text field is associated.

Figure 7-19 Connect widgets with associations



Associate each widget appropriately. Save the nib file.

Build and Run

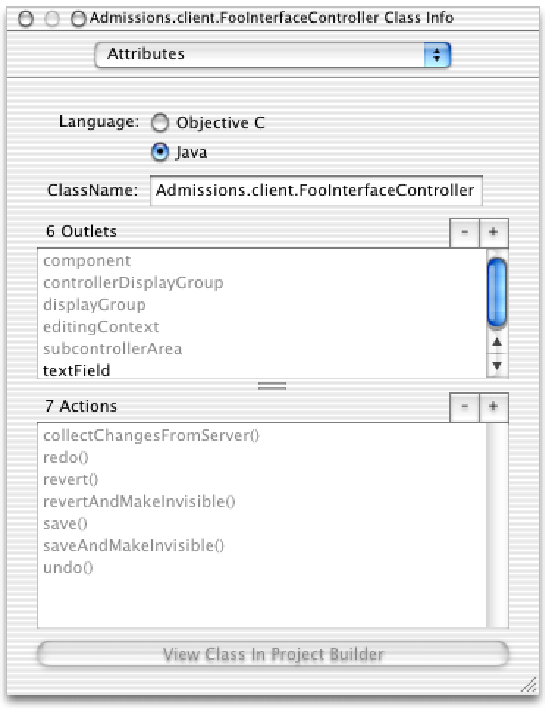
In Project Builder, build and run the application just as you would for a Direct to Java Client application.

Programmatic Access to Interface Components

It's common to want programmatic access to user interface components in nib files. In the Cocoa application framework, outlets for all appropriate user interface components are added to the corresponding Java file for that interface upon adding a component. However, this doesn't happen when building Java Client interfaces in Interface Builder. Fortunately, it's easy to add this functionality to your application.

In a nib file, select File's Owner in the nib file window, switch to the Classes pane, and choose Add Outlet from the Classes menu. If the widget in question is a text field, for instance, name the outlet `textField` as shown in Figure 7-20 (page 152).

Figure 7-20 Add an outlet


Admissions.client.FooInterfaceController Class Info
Attributes
View Class In Project Builder

Language: [ ] Objective C [✓ Java]

ClassName: Admissions.client.FooInterfaceController

6 Outlets

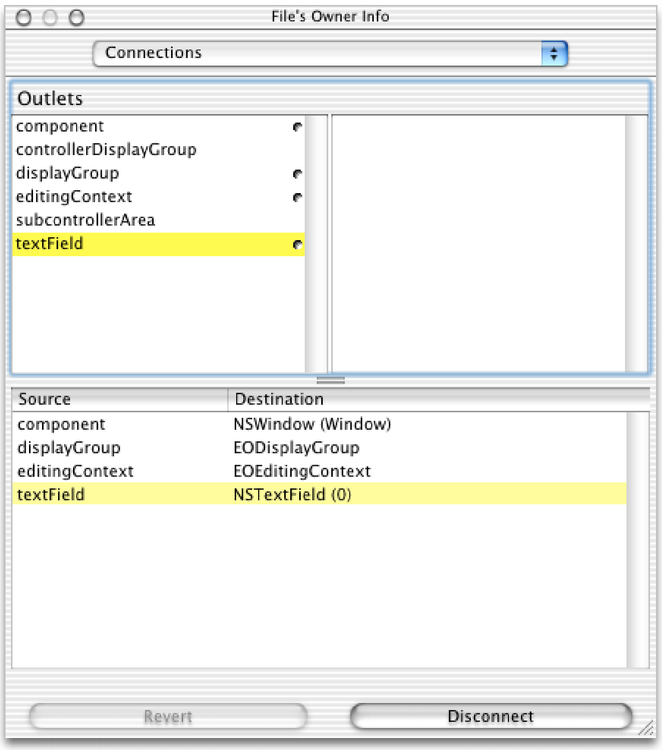
component
controllerDisplayGroup
displayGroup
editingContext
subcontrollerArea
textField

7 Actions

collectChangesFromServer()
redo()
revert()
revertAndMakeInvisible()
save()
saveAndMakeInvisible()
undo()

Connect the new outlet to a text field by Control-dragging from the File’s Owner icon to the text field and double-clicking `textField` in the Outlets column of the Connections pane of the File’s Owner Info window, as shown in Figure 7-21 (page 153).

Figure 7-21 Connect the new outlet



Source	Destination
component	NSWindow (Window)
displayGroup	EODisplayGroup
editingContext	EOEditingContext
textField	NSTextField (0)

In the Java class file for the nib file, add a public variable called `textField`. You now have programmatic access to the value and attributes of that widget. This is useful, for instance, if you want to manipulate or extract the values of a particular widget.

Cocoa to Swing Translation

In nondirect Java Client applications, you use Interface Builder to construct user interfaces. This application was intended to build Mac OS X Cocoa or Carbon applications. It was not designed to build Swing-based applications, and in fact, it does not build them directly. Rather, technology in Java Client translates Interface Builder Cocoa nib files to Swing for you.

Cocoa offers many user interface widgets, and Java Client supports most of them. Certain widgets are not supported because there is no Swing equivalent. Java Client translates the following user interface elements:

- **Cocoa widgets:** NSWindow, NSButton, NSTextField, NSTextView, NSTableView, NSTableColumn, NSComboBox, NSPopUpButton, NSMatrix, NSForm, NSBox, NSImageView, NSTabView, NSCustomView; the corresponding cells for these widgets are also translated
- **Formatters:** NSNumberFormatter, NSTimestampFormatter
- **Enterprise Objects:** EOEditingContext, EODisplayGroup, EODataSource, EOAssociation
- **Interface Builder connections:** Outlets, target-action, `nextKeyView`

- **Style attributes:** Font sizes and some font styles (font styles depend on Swing's ability to find and load fonts)

Java Client interface translation does not support colors, menus, scroll views (except when in text or table views), NSBrowsers, NSOutlineViews, NSProgressIndicators.

If you want to use a widget that is not supported by the Java Client interface translation, you can use an NSCustomView object and associate it with a Swing class you write. This is described in "Using Custom Views in Nib Files" (page 213).

Building Custom Controllers With XML

Direct to Java Client's dynamic user-interface generation may not always provide all the kinds of user interfaces your applications require. As described in "Nondirect Java Client Development" (page 137), you can use Interface Builder to build nondirect, static interfaces and use them within Direct to Java Client applications. Interface Builder generates a Java class file that contains Swing code. Although this saves you from writing Swing, it is not ideal.

When you use nib files, you need to provide a separate nib file for each language and platform your application supports. So if your customers use both Windows and Mac OS X and require the application to be localized in French, German, and English, you need to build at least six nib files for every nib file your application uses. The maintenance burden should be obvious.

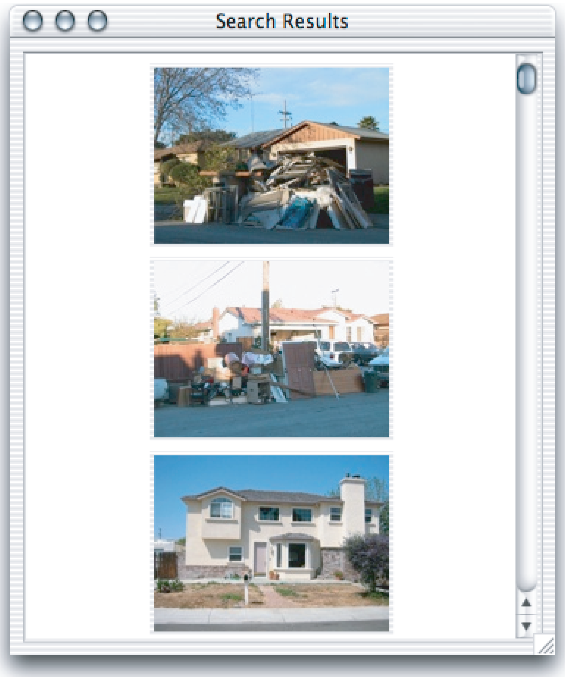
When you use controller classes, however, you need to write only a single version. Direct to Java Client dynamically provides localized and platform-specific versions of the controller when requested by the client. And when you need to write custom controllers for user interface elements that are not provided by Direct to Java Client, you need to write only minimal Swing code.

This chapter describes both how to write a custom controller class for a user interface element that isn't provided by Direct to Java Client and also how to define user interfaces in XML using the provided controllers.

Custom Controller Class

When you run the JCRealEstatePhotos example, you may recognize a custom controller—the scroll view controller that displays photos of listings that are returned by a query. This controller is shown in Figure 8-1.

Figure 8-1 Scroll view controller with clickable images



It was necessary to write a custom controller class to achieve this functionality because Direct to Java Client does not provide a scroll view controller whose contents are images. Direct to Java Client provides a number of prebuilt controller classes, but your applications may need to use specialized controllers that it doesn't provide.

The custom scroll view controller requires you to do three things: write a Java class file that defines the controller in Swing, add a D2WComponent to the project that contains XML that references the custom controller, and register a new task with the rule system that can be invoked to display the custom controller.

Java Swing Class

Custom controller classes that are used to provide widget-level functionality to an application inherit from `com.webobjects.eogeneration.EOWidgetController`. All you need to do to provide a custom widget is to subclass that class and override the `newWidget` method, which returns a JComponent. The ScrollViewController class in the JCRealEstatePhotos example constructs and returns a JPanel that contains a JScrollPane that contains a number of JButton subclassed objects. See the class file in the example project.

XML Hierarchy

The ScrollViewController class is referenced in a D2WComponent's XML description—that's how it gets displayed. That XML description is quite simple, as shown in Listing 8-1. The JCRealEstatePhotos example names this component ScrollViewComponent.

Listing 8-1 Reference the custom controller in XML

```
<FRAMECONTROLLER reuseMode="never" horizontallyResizable="false"
       verticallyResizable="true" label="Search Results" disposeIfDeactivated="true">
    <CONTROLLER
         className="webobjectsexamples.realestatephotos.client.ScrollViewController"
          horizontallyResizable="true" verticallyResizable="true" minimumHeight="300"
          minimumWidth="300"/>
</FRAMECONTROLLER>
```

The other controllers in the XML description give you a lot of functionality for free. The `FRAMECONTROLLER` controller displays a JFrame, gives it a label, and specifies its resizing constraints. Likewise, the `CONTROLLER` controller references the custom controller class and adds it to the JFrame specified by the `FRAMECONTROLLER` controller, specifies its resizing constraints, and specifies its minimum dimensions. This is one of the great benefits of controllers—you can write user interfaces at a higher level than raw Swing.

The Rule

To actually display the custom controller, you need to invoke a particular task. So, you need to first register a new task with the rule system. The JCRealEstatePhotos example uses this rule to register a new task that displays the custom controller:

Left-Hand Side:	`(task='scroll')`
Key:	`window`
Value:	`"ScrollViewComponent"`
Priority:	`50`

You can ask the controller factory to invoke the task with this call:

```
EOControllerFactory.sharedControllerFactory().openWindowForTaskName("scroll");
```

That's all you need to do to write and use static controllers that are not based on nib files in a Direct to Java Client application. Although writing your own controller hierarchies may not be fun, it's the right way to build a static user interface for use in a Direct to Java Client application. You can use the Direct to Java Client Assistant to get clues on how to structure the XML hierarchy to get the user interface you want.

Use Controllers Rather Than Nib Files

Building user interfaces in Interface Builder allows you to specify exactly the layout of a particular window. With that tool, you can specify exact widget sizes, the spacing between widgets, and other particular characteristics of a user interface. This is the advantage of using nib files: You have exact control over the look of a user interface.

However, you must weigh this advantage against the maintenance costs of using nib files. A good compromise between the precise user interface layout you can achieve in Interface Builder and dynamically generated user interfaces that you get for free from Direct to Java Client is to build up a user interface controller hierarchy by hand. While this does not afford you quite the same flexibility as using a nib file, you should able to achieve similar results.

Figure 8-2 shows a user interface built in Interface Builder. A similar user interface built with a controller hierarchy appears in Figure 8-3.

Figure 8-2 Nib-based query window

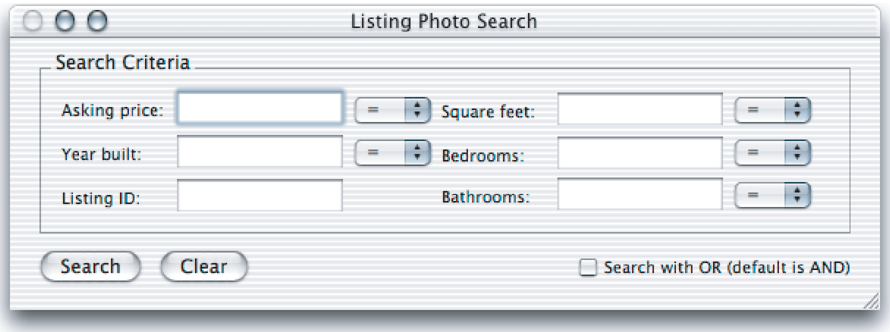
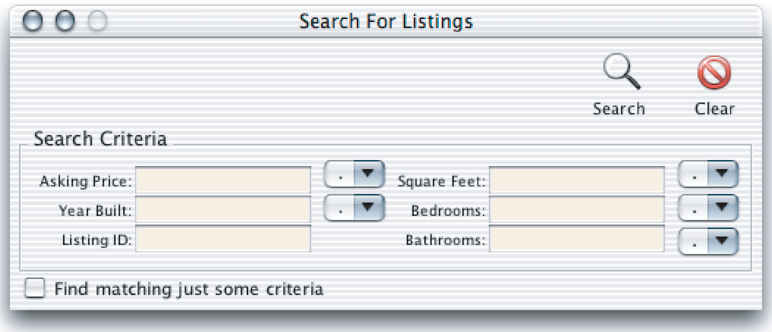


Figure 8-3 Controller-based query window



The controller hierarchy for the user interface in Figure 8-3 looks like Listing 8-2.

Listing 8-2 Controller hierarchy for a user interface

```
<FRAMECONTROLLER horizontallyResizable="false" verticallyResizable="false" label="Search
 For Listings" minimumWidth="550">
 <ACTIONBUTTONSCONTROLLER widgetPosition="TopRight">
  <CONTROLLER usesHorizontalLayout="false" verticallyResizable="false"
     className="webobjectsexamples.realestatephotos.client.SearchController">
    <BOXCONTROLLER borderType="Etched" usesTitledBorder="true" label="Search
       Criteria" usesHorizontalLayout="true" horizontallyResizable="true"
       verticallyResizable="false">
      <BOXCONTROLLER usesTitledBorder="false" alignsComponents="true" borderType="None"
        horizontallyResizable="true" verticallyResizable="false">
       <TEXTFIELDCONTROLLER suppressesAssociation="true" label="Asking Price"
         typeName="askingPrice input field" isQueryWidget="true"/>
       <TEXTFIELDCONTROLLER suppressesAssociation="true" label="Year Built"
         typeName="yearBuilt input field" isQueryWidget="true"/>
       <TEXTFIELDCONTROLLER suppressesAssociation="true" label="Listing ID"
         typeName="listingID input field" isQueryWidget="true"/>
     </BOXCONTROLLER>
     <COMPONENTCONTROLLER alignsComponents="true" horizontallyResizable="false"
```

```
          verticallyResizable="false">
         <CONTROLLER
            className="webobjectsexamples.realestatephotos.client.ComboBoxController"
            typeName="askingPrice combo box" minimumWidth="55" minimumHeight="22"/>
         <CONTROLLER
            className="webobjectsexamples.realestatephotos.client.ComboBoxController"
            typeName="yearBuilt combo box" minimumWidth="55" minimumHeight="22"/>
        </COMPONENTCONTROLLER>
       <BOXCONTROLLER usesTitledBorder="false" alignsComponents="true" borderType="None"
          horizontallyResizable="true" verticallyResizable="false">
         <TEXTFIELDCONTROLLER suppressesAssociation="true" label="Square Feet"
           typeName="squareFt input field" isQueryWidget="true"/>
         <TEXTFIELDCONTROLLER suppressesAssociation="true" label="Bedrooms"
            typeName="bedrooms input field" isQueryWidget="true"/>
         <TEXTFIELDCONTROLLER suppressesAssociation="true" label="Bathrooms"
            typeName="bathrooms input field" isQueryWidget="true"/>
         </BOXCONTROLLER>
           <COMPONENTCONTROLLER alignsComponents="true" horizontallyResizable="false"
              verticallyResizable="false">
              <CONTROLLER
               className="webobjectsexamples.realestatephotos.client.ComboBoxController"
                 widgetPosition="Right" typeName="squareFt combo box" minimumWidth="55"
                 minimumHeight="22"/>
              <CONTROLLER
               className="webobjectsexamples.realestatephotos.client.ComboBoxController"
                 widgetPosition="Right" typeName="bedrooms combo box" minimumWidth="55"
                 minimumHeight="22"/>
               <CONTROLLER className="webobjectsexamples.realestatephotos.client.ComboBo
                xController" widgetPosition="Right" typeName="bathrooms combo box"
                 minimumWidth="55" minimumHeight="22"/>
           </COMPONENTCONTROLLER>
          </BOXCONTROLLER>
          <CHECKBOXCONTROLLER alignment="Right" suppressesAssociation="true"
          typeName="searchtype input checkbox" label="Find matching just some criteria"/>

          </CONTROLLER>
      </ACTIONBUTTONSCONTROLLER>
  </FRAMECONTROLLER>
```

There are a few interesting things to point out in Listing 8-2. Except for the window frame and button toolbar, the rest of the controllers are wrapped in a `CONTROLLER` tag whose class is `webobjectsexamples.realestatephotos.client.SearchController`. This class inherits from EOComponentController and is shown in Listing 8-3 (page 160). It serves as the supercontroller for the controllers nested within it. A supercontroller has access to all the controllers nested within it, which allows you to retrieve information about those nested controllers and any data they contain (such as text users enter in a text field or the value of a checkbox).

Most of the controllers in Listing 8-2 include the attribute `suppressesAssociation="true"`. This tells the rule system that a particular controller is not associated with an enterprise object. Rather, that controller is just a user interface controller that is used to display a user interface widget and allows a user to enter data.

All the controllers in Listing 8-2 that accept user data include a `typeName` tag. This tag is used to identify each controller in the controller hierarchy so when you traverse the controller hierarchy looking for data, you can identify specific controllers.

Retrieving Values From a Controller Hierarchy

One of the tasks you need to perform when using a controller-based user interface is getting values the user enters. As discussed in "Use Controllers Rather Than Nib Files" (page 157) and as implemented in Listing 8-2 (page 158), by assigning controllers a value for the `typeName` attribute, you can traverse a given controller hierarchy and look up controllers based on that attribute.

The methods `valueFromInputField`, `valueFromCheckBox`, and `valueFromComboBox` in the SearchController class in Listing 8-3 accomplish this. Remember, SearchController is the supercontroller for the user interface elements in the controller hierarchy in Listing 8-2 (page 158). From any supercontroller, you can traverse its subcontroller hierarchy to find certain controllers. This is what the `valueFrom` methods in Listing 8-3 do.

Listing 8-3 SearchController class

```
package webobjectsexamples.realestatephotos.client;

import com.webobjects.foundation.*;
import com.webobjects.eoapplication.*;
import com.webobjects.eogeneration.*;
import javax.swing.*;
import java.awt.*;

public class SearchController extends EOComponentController {

    public SearchController(EOXMLUnarchiver unarchiver) {
        super(unarchiver);
    }

    protected NSArray defaultActions() {
        NSMutableArray actions = new NSMutableArray();

        String title = EOUserInterfaceParameters.localizedString("Search");
        String clearTitle = EOUserInterfaceParameters.localizedString("Clear");

        Icon icon = EOUserInterfaceParameters.localizedIcon("ActionIconFind");
       actions.addObject(EOAction.actionForControllerHierarchy("search", title, title,
         icon, null, null, 0, 0, false));

        icon = EOUserInterfaceParameters.localizedIcon("ActionIconClear");

        actions.addObject(EOAction.actionForControllerHierarchy("clear", clearTitle,
           clearTitle, icon, null, null, 0, 300, false));
        return EOAction.mergedActions(actions, super.defaultActions());
    }

    public void search() {

        NSMutableDictionary valuesAndSuch = new NSMutableDictionary();
        valuesAndSuch.takeValueForKey(valueFromInputField("askingPrice"),
           "askingPriceValue");
        valuesAndSuch.takeValueForKey(valueFromComboBox("askingPrice"),
           "askingPriceCompareBy");
        valuesAndSuch.takeValueForKey(valueFromInputField("yearBuilt"),
           "yearBuiltValue");
        valuesAndSuch.takeValueForKey(valueFromComboBox("yearBuilt"),
           "yearBuiltCompareBy");
        valuesAndSuch.takeValueForKey(valueFromInputField("listingID"),
           "listingIDValue");
        valuesAndSuch.takeValueForKey(valueFromInputField("squareFt"),
           "squareFtValue");
        valuesAndSuch.takeValueForKey(valueFromComboBox("squareFt"),
           "squareFtCompareBy");
        valuesAndSuch.takeValueForKey(valueFromInputField("bedrooms"),
           "bedroomsValue");
        valuesAndSuch.takeValueForKey(valueFromComboBox("bedrooms"),
           "bedroomsCompareBy");
        valuesAndSuch.takeValueForKey(valueFromInputField("bathrooms"),
           "bathroomsValue");
        valuesAndSuch.takeValueForKey(valueFromComboBox("bathrooms"),
           "bathroomsCompareBy");
        valuesAndSuch.takeValueForKey(valueFromCheckBox("searchtype"), "searchtype");

        ImageSearch search = new ImageSearch(valuesAndSuch);
    }

    public String valueFromInputField(String identifier) {
        EOWidgetController controller =
         (EOWidgetController)(controllerWithKeyValuePair(EOController.SubcontrollersEnu
         meration, null, "typeName", identifier + " input field"));
        if (controller == null) {
            throw new IllegalStateException("Can't find input checkbox controller for
"             + identifier);
        }
        return (((JTextField)controller.widget()).getText());
    }

    public String valueFromComboBox(String identifier) {
        EOWidgetController controller =
          (EOWidgetController)(controllerWithKeyValuePair(EOController.SubcontrollersEn
           umeration, null, "typeName", identifier + " combo box"));
        if (controller == null) {
            throw new IllegalStateException("Can't find input checkbox controller for
"

              + identifier);
        }
        return (String)(((JComboBox)controller.widget()).getSelectedItem());
    }

    public Boolean valueFromCheckBox(String identifier) {
        EOWidgetController controller =
         (EOWidgetController)(controllerWithKeyValuePair(EOController.SubcontrollersEnu
         meration, null, "typeName", identifier + " input checkbox"));
        if (controller == null) {
            throw new IllegalStateException("Can't find input checkbox controller for
"

           + identifier);
        }
        return new Boolean(((JCheckBox)controller.widget()).isSelected());
    }

    public void clear() {
        NSLog.out.appendln("clearing in controller");
        }

}
```

The `search` method collects the search criteria as entered by the user, as well as the values of the combo boxes that control how the qualifier for a particular search criteria is constructed. It then passes these values in a dictionary to the ImageSearch class (not shown here), which actually performs the search.

Inside the Rule System

The rule system is responsible for analyzing EOModels and from them, generating the XML for dynamic user interfaces. It answers the following questions:

- What kind of windows are available?
- Which of these windows should open at application startup?
- What kind of actions should be presented to the user?
- What should happen when a particular condition in an application occurs?

How It Works

From the answers to these questions, the rule system builds a detailed description of the user interface. When the controller factory sends a request to the server application, the rule system works with a set of `com.webobjects.directtoweb.D2WComponent` classes and WebObjects dynamic elements to generate the XML. The rule system receives the controller factory's requests and evaluates rules to determine which D2WComponent subclasses should generate the XML for the current request. The D2WComponent subclasses (using WOXMLNode dynamic elements internally) perform the actual XML generation.

All the information about how to configure a Direct to Java Client application is stored in rule system rules. A rule has a key, a value, and a priority. A key is a condition that must be true for the rule to fire. The rule system evaluates requests as follows:

1. The controller factory makes a request to the rule system by specifying a key.
2. The rule system identifies the rules whose key is the same as the request key.
3. It then evaluates the conditions of the matching rules to see which can fire.
4. Of the rules that can fire, the rule system fires the one with the highest priority, returning the value for the rule's key.

By specifying `-D2WTraceRuleFiringEnabled YES` as a launch argument on the server application, you can see all the rules fire in a Direct to Java Client application.

To evaluate requests, the rule system needs information about the state of the client application. In addition to specifying a key, the controller factory also provides key-value pairs of state information that the rule system can use to evaluate the conditions of rules. For example, the rule system might need to know what task the client application is attempting to perform (query, list, or form) and the entity on which the client application is operating.

The controller factory packages all rule system input—the request key and the key-value pairs of state information—into a dictionary known as a specification. The following are examples of specifications:

- `question = window, task = query`
- `question = window, entity.name = Student, task = form`
- `question = modalDialog, entity.name = Activity, task = select`

A specification always contains a question, which is the request key. The request keys in the above examples are `window` and `modalDialog`.

In the rule system, you have access to the user's language and platform, so you can write rules to provide application behavior based on those attributes of the client. This allows for mostly automatic localization of rule-generated components (as described in "Localizing Dynamic Components" (page 235)) and provides automatic platform-specific user interface layout.

The rule system stores the key-value pairs of state information in a `com.webobjects.directtoweb.D2WContext` object. The D2WContext's whole purpose is to keep track of state as a response is generated. Initially the D2WContext contains state information provided by the controller factory. As the rule system processes requests, the system adds more state information to the D2WContext.

Rule System Priorities

Each rule system rule has a priority, which is a mechanism to manage conflicting rules. It is possible to have two rules with the same condition (left-hand side), the same key (right-hand side), but a different value (right-hand side). To handle this conflict, the rule with the higher priority is fired.

The default rules provided by the `com.webobjects.eogeneration` package have a priority of 0. The Direct to Java Client Assistant gives its rules a priority of 100. You should never change the priority of rules generated by Assistant (this would involve editing the `user.d2wmodel` file, which is never a good idea since Assistant writes it out each time it saves). Also, the rules you create by hand should not have a priority of 100, since this confuses Assistant. If you want the rules created by Assistant to be preferred over your own rules, use a lower priority like 50. Otherwise, give your rules a higher priority.

D2WComponents

There is a one-to-one correspondence between Direct to Java Client D2WComponent subclasses (server side) and EOController subclasses (client side). For example, an EOTextFieldController (inheriting from EOController) on the client has a corresponding D2WComponent class on the server also named EOTextFieldController (inheriting from D2WComponent). The client-side class displays and manages user interface widgets while the server-side class generates XML to describe the client-side user interface.

The server-side D2WComponents for Direct to Java Client applications can be found in `/System/Library/Frameworks/JavaEOGeneration.framework/Resources`. You can open the components in WebObjects Builder to learn more about them.

Rule System Requests

The user interface components of a Direct to Java Client application are generated by the rule system when needed. The controller factory makes rule system requests as each new window in the client application is activated.

When an application starts up, the controller factory makes requests for the following keys:

- `availableSpecifications`, which tells the controller factory all the specifications (request keys such as `window` and `modalDialog`, and any custom request keys)
- `defaultSpecifications`, which tells the controller factory which windows to open automatically once the application is finished initializing
- `actions`, which tells the controller factory what actions to add to the main menu along with standard menu items such as Quit

Then, to generate the controller hierarchy for a window or modal dialog, the controller factory makes requests for the following keys:

- `window`, which returns the controller hierarchy XML for a window the application will open; the request from the controller hierarchy must also provide state information such as a task and, optionally, an entity name so the rule system can determine what window is being generated
- `modalDialog` , which returns the controller hierarchy XML for a modal dialog the application will open; again, the request from the controller hierarchy must also provide state information such as a task and, optionally, an entity name

Internal Rule System Requests

When the rule system evaluates a request from the controller factory, the actual returned value is the name of a D2WComponent, not the controller hierarchy XML. The D2WComponent identified by the fired rule is responsible for generating the controller hierarchy XML that the controller factory receives.

In the process of generating XML, the D2WComponent objects might require the rule system to evaluate additional requests, the most significant of which are these two:

- `controller`, which identifies a controller (a D2WComponent) for a task identified in the request's specification; the entity-level controller defines the part of a window or dialog user interface for performing the specified task on the specified entity
- `propertyKeys`, which identifies the property keys for a task and an entity identified in the request's specification; the property keys are needed to identify the additional controllers needed to display and manipulate an object's attributes and relationships

Sometimes it is necessary to know what kind of controller the rule system asks for. All the default rules therefore put additional information on the D2WContext (and you should maintain this information if you customize rules). This information can be used as additional criteria in the rule qualifiers. These are the two categories of controllers:

- `controllerType` , (possible values: `actionWidgetController`, `dividingController`, `groupingController`, `entityController`, `modalDialogController`, `tableController`, `widgetController`, `windowController`)
- `isRootController`, (`false` if not, `nil` otherwise)

Generating the Student Form Window

As an example of how the Direct to Java Client D2WComponent classes work, consider the form window for the Student entity in the tutorials. Suppose a user clicks the New button in a Query window for the Student entity. The controller factory then makes a request to the rule system with the following specification:

```
question = window, entity.name = Student, task = form
```

This specification tells the rule system that for the `form` task for the `Student` entity it should evaluate the `window` key and return a controller hierarchy based on what the `window` key evaluates to in the rule system.

The default rule fired to satisfy this request is as follows:

Left-Hand Side:	`*true*`
Key:	`window`
Value:	`"EOWindow"`
Priority:	`0`

You could write a custom rule (and give it a higher priority) to, for example, associate an EOModalDialog with the `window` key rather than the default EOWindow. Since in this case the default rule is not overridden, the D2WComponent that generates the XML for the form task for the Student entity is EOWindow (a WebObjects component).

Open `EOWindow.wo` in WebObjects Builder. (You can find it in `/System/Library/Frameworks/JavaEOGeneration.framework/Resources`).

`EOWindow.wo` contains a `.html` file (containing XML) and a `.wod` file.

Here's an excerpt from `EOWindow.html`:

```
    <WEBOBJECT name=windowController>
        <WEBOBJECT name=actionWidgetController>
            <WEBOBJECT name=taskController>
            </WEBOBJECT>
        </WEBOBJECT>
        <WEBOBJECT name=content>
        </WEBOBJECT>
    </WEBOBJECT>
```

And here's an excerpt from `EOModalDialog.wod`:

```
windowController: EOSwitchComponent {
    componentNameKey = "windowController";
    d2wContext = localContext;
    controllerType = "windowController";
}
```

The EOSwitchComponent in the `.wod` file is a dynamic element that makes a new rule system request using the `componentNameKey` as the request key. So in the case of `windowController`, the switch component makes a new rule system request with the key `windowController`, the name of the `componentNameKey` binding.

Before making the request, however, the switch component updates the rule system's state information. Generally it creates a new D2WContext based on the state information in the old D2WContext. That's what the `d2wcontext` binding specifies. Bindings other than `componentNameKey` and `d2wcontext` identify additional state that the switch component adds to the new D2WContext. For `windowController`, the additional state is simply that the `controllerType` is `windowController`.

In this manner, the XML controller hierarchy is built recursively using switch components.

One of the leaf nodes in the Student form window is for an EOFormController whose `.wod` file looks like this:

```
content: WOComponentContent {
}

controller: WOXMLNode {
    elementName = "FORMCONTROLLER";
    alignmentWidth = d2wContext.alignmentWidth;
    alignsComponents = d2wContext.alignsComponents;
    archive = d2wContext.archive;
    className = d2wContext.className;
    displayGroupProviderMethodName =
d2wContext.displayGroupProviderMethodName;
    editability = d2wContext.editability;
    editingContextProviderMethodName =
d2wContext.editingContextProviderMethodName;
    entity = controllerEntityName;
    horizontallyResizable = d2wContext.horizontallyResizable;
    iconName = d2wContext.iconName;
    iconURL = d2wContext.iconURL;
    label = d2wContext.label;
    minimumHeight = d2wContext.minimumHeight;
    minimumWidth = d2wContext.minimumWidth;
    path = controllerRelationshipPath;
    prefersIconOnly = d2wContext.prefersIconOnly;
    transient = d2wContext.transient;
    usesHorizontalLayout = d2wContext.usesHorizontalLayout;
    verticallyResizable = d2wContext.verticallyResizable;
}
disabledActionNamesArray: EOSwitchComponent {
    componentName = "EOStringArray";
    array = d2wContext.disabledActionNames;
    name = "disabledActionNames";
}
mandatoryRelationshipPathsArray: EOSwitchComponent {
    componentName = "EOStringArray";
    array = d2wContext.mandatoryRelationshipPaths;
    name = "mandatoryRelationshipPaths";
}
```

A WOXMLNode is a component that generates XML for a node in the controller hierarchy. Its bindings tell the server-side D2WComponent how to configure its client-side counterpart. For example, the binding names in the EOFormController `.wod` file correspond to XML attributes understood by the client-side EOFormController. Correspondingly, the binding values are the values assigned to those XML attributes. Most of the bindings are set to a key path starting with "d2wContext". These key paths refer to the state information stored in the D2WContext.

EOSwitchComponent

EOSwitchComponent is a special dynamic element that takes a D2WContext and passes it as a copy with additional arguments to a D2WComponent. Usually it is used to pass a D2WContext to a subcomponent, but since the context is copied first, the context of the parent component is not modified and can be passed to other subcomponents without risk.

There are three bindings on EOSwitchComponent:

- `componentName`—name of the D2WComponent to evaluate
- `componentNameKey`—gets the name of the D2WComponent from a key
- `d2wContext`—the D2WContext to copy

The `componentName` and `componentNameKey` bindings are mutually exclusive (only one of them can be used). `d2wContext` is usually `localContext` (usually the EOSwitchComponent is used inside a D2WComponent and `localContext` returns the D2WContext of it then).

All other bindings on the EOSwitchComponent are considered additional parameters for the newly created D2WContext.

Example:

```
queryListController: EOSwitchComponent {
    componentNameKey = "controller";
    d2wContext = localContext;
    controllerType = noValue;
    forceHorizontallyNotResizable = noValue;
    forceVerticallyNotResizable = noValue;
    forceEntityReadOnly = "true";
    forceWidgetReadOnly = "true";
    isRootController = "false";
    propertyKey = noValue;
    task = "list";
}
```

This creates a D2WContext with the local context of the component using this entry in the `.wod` file, gets the name of the contained component from `controller`, and adds all the other values to the D2WContext passed to that component.

Deploying Client Applications

There are two phases in deploying Java Client applications: deploying the server-side application and deploying the client-side application. If you're familiar with deploying HTML-based WebObjects applications, you already know everything you need to deploy the server-side application. If you're unfamiliar with this process, however, you need to read the document *WebObjects Deployment Guide Using JavaMonitor*.

After you've deployed the server-side part of a Java Client application, there are two ways you can deploy the client application. They are described in this chapter, after a section comparing the three options:

Deployment Options

HTML-based WebObjects applications require only a Web browser on the client to run on the user's computer. The client requirements for Java Client desktop applications, however, are considerably more demanding.

Java Client applications can be deployed as real desktop applications or through Web Start. Each deployment option is feasible, but you should carefully evaluate both options after understanding their respective strengths and weaknesses.

- **Installation:** Applets and Web Start require no installation of the Java Client application on the user's part, since the Web browser or Web Start application handles the downloading of classes. Applications, however, need to be installed on the client.
- **Upgrades:** Using applets or Web Start, the upgrade process is invisible to the user. Using applications, the user must perform upgrades manually, and a versioning scheme must be devised to ensure compatibility between client and server.
- **Platform support:** All deployment options require the presence of JRE 1.3 or later on the client. Mac OS X provides out-of-the-box support for JRE 1.3. On other platforms, the JRE must be downloaded and installed. Internet Explorer for Mac OS X 10.1 supports embedded applets as well, and Sun's Java plug-in for Web browsers provides support for running applets on other platforms. So, in terms of portability, systems with the correct JRE can run Java Client applications as applets, as full desktop applications, or through Web Start.
- **User experience:** Running Java Client applications as desktop applications or through Web Start always provides a better user experience than running as applets. Applets can take down the Java virtual machine and other applets, and applets generally don't have the fit and finish of Java desktop applications or Web Start applications. Java Client applications running as desktop applications or as Web Start applications in Mac OS X take advantage of platform-specific features such as the global menu bar and the dirty window marker.

- **Performance:** Generally, applications perform better than applets and Web Start applications, since Web browsers provide more security checks than applications and perform other operations that degrade performance. But even so, the performance difference between applets and applications or Web Start applications should be an insignificant factor in choosing a deployment method.
- **Security:** From the user's perspective, running as applets or through Web Start is inherently more secure, since those deployment options prevent the JVM from accessing the file system or other parts of the user's system. Web Start allows applications to access the user's system only after the user authorizes such access. Developers usually prefer applications over applets and Web Start because they don't have to worry about the security restrictions inherent to Web browsers, but this usually isn't a deciding factor when choosing how to deploy the client-side application.

Web Start

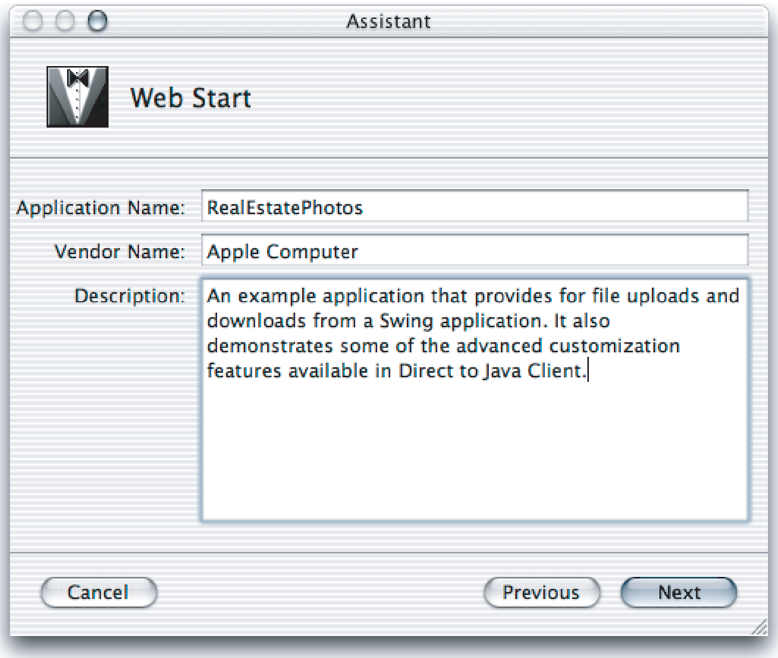
The easiest way to deploy the client part of Java Client applications is to use Web Start. Starting with WebObjects 5.2, Web Start is integrated with all Java Client applications, which takes care of these Web Start tasks for you:

- writing a JNLP file
- creating `.jar` files from `.class` files

In WebObjects 5.2, Java Client projects have two default WOComponent files, `Main.wo` and `JavaClient.wo`. `Main.wo` provides an HTML entry page for the client application that includes an HTML link to the JNLP file of the client application. The JNLP file is generated dynamically when the application starts up so you don't need to worry about it at all.

Attributes of the JNLP file such as vendor, description, and application name are specified with bindings in the JavaClient component. You probably configured these bindings in the Java Client project assistant, as shown in Figure 10-1.

Figure 10-1 JNLP configuration in new project assistant



Application Name: RealEstatePhotos
Vendor Name: Apple Computer
Description: An example application that provides for file uploads and downloads from a Swing application. It also demonstrates some of the advanced customization features available in Direct to Java Client.

When a user clicks the link that points to the client application's JNLP file, that file is downloaded to their computer. If you've added the JNLP MIME type to your Web server, the client's Web browser should automatically invoke the Web Start application to launch the client application.

Note: If you're deploying the application on Apache, you register the JNLP MIME type by adding this line to the `mime.types` file (in Mac OS X, this file is in `/etc/httpd/mime.types`):
```
application/x-java-jnlp-file jnlp
```

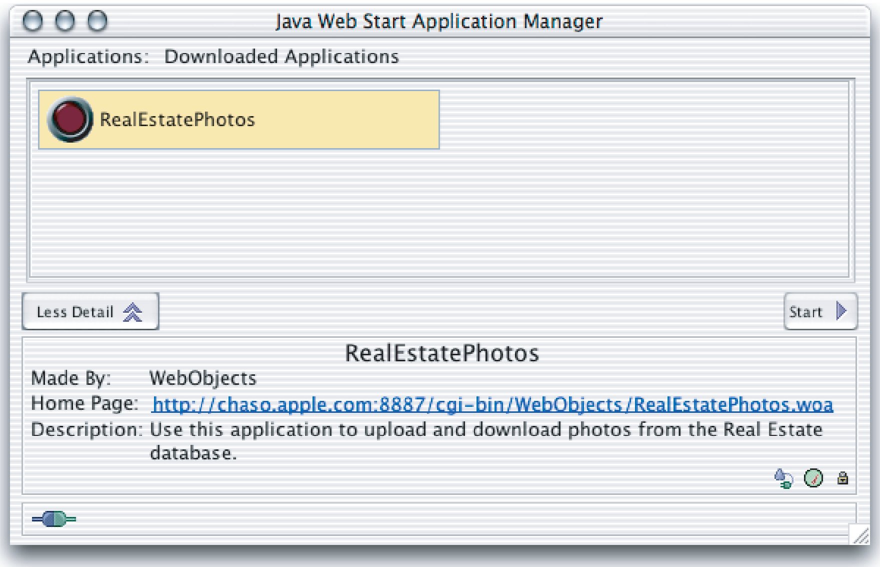
Deploying the client application with Web Start provides a platform-independent, standards-based deployment solution that is easy for end users to use and that is easy for you to update and maintain. It is perfect for an environment of distributed heterogeneous client systems.

Java Client signs the `wojavaclient.jar` file (the core Java Client client-side runtime classes) with its own security certificate. However, there is no way you or your clients can verify the authenticity of this certificate, so the `wojavaclient.jar` appears as signed by an "unknown issuer" and recommend to users that they do not install applications that use it. If this presents a problem for your organization, you'll need to resign the `wojavaclient.jar`. If your application includes custom client-side classes (such as nib files and custom controller classes), you may also need to sign the `.jar` file that contains them with your certificate.

With that said, Java Client applications usually do not need to be run as trusted since they don't usually need access to the client computer's file system.

To make accessing your application easier for users, you should instruct them to use the Web Start application to launch the client application after the first use. This frees them from needing to remember the Web address of the client application. Figure 10-2 shows the Web Start application running in Mac OS X.

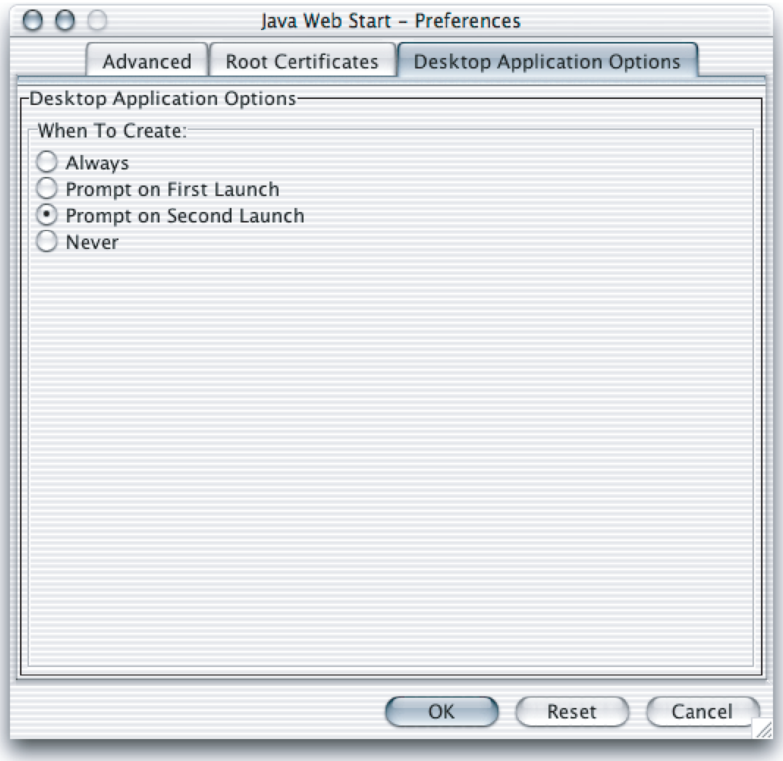
Figure 10-2 Run the client application from the Web Start application


Java Web Start Application Manager
Applications: Downloaded Applications
RealEstatePhotos
Less Detail
Start

RealEstatePhotos
Made By: WebObjects
Home Page: http://chaso.apple.com:8887/cgi-bin/WebObjects/RealEstatePhotos.woa
Description: Use this application to upload and download photos from the Real Estate database.

The Web Start application also includes a preference to make a platform-specific application executable on the user's computer so that they can launch the application without needing to use the Web Start application or visit the client application's Web page. This preference pane is shown in Figure 10-3.

Figure 10-3 Configure Web Start to create a desktop application for the client application



Desktop Application Options
When To Create:
- Always
- Prompt on First Launch
- [✓ Prompt on Second Launch]
- Never

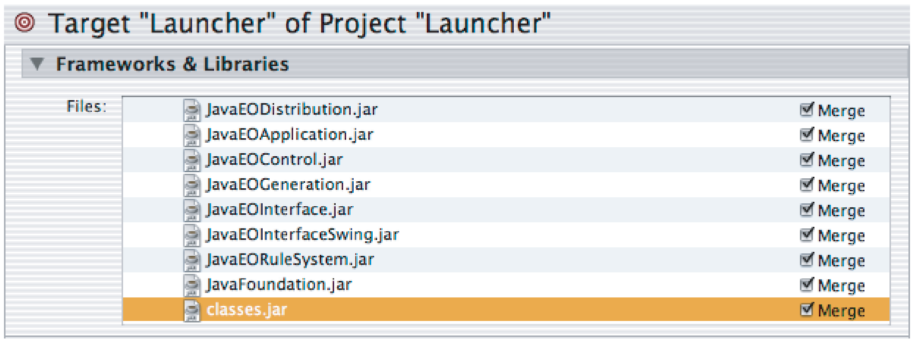
Desktop Applications

Although Web Start is the default and recommended client deployment mechanism, you can also deploy the client part of Java Client applications as platform-specific desktop applications. You may want to do this to avoid the security contract imposed by Web Start or to achieve optimum performance.

To do this in Mac OS X, follow these steps in Project Builder:

1. Make a new project of type Java Swing Application.
2. Add all the `.jar` files for the frameworks the client application uses. These include at least the following:
   - `/System/Library/Frameworks/JavaEOApplication.framework/WebServerResources/JavaEOApplication.jar`
   - `/System/Library/Frameworks/JavaEOControl.framework/WebServerResources/JavaEOControl.jar`
   - `/System/Library/Frameworks/JavaEODistribution.framework/WebServerResources/JavaEODistribution.jar`
   - `/System/Library/Frameworks/JavaFoundation.framework/WebServerResources/JavaFoundation.jar`
   - `/System/Library/Frameworks/JavaEOGeneration.framework/WebServerResources/JavaEOGeneration.jar`
   - `/System/Library/Frameworks/JavaEOInterface.framework/WebServerResources/JavaEOInterface.jar`
   - `/System/Library/Frameworks/JavaEOInterfaceSwing.framework/WebServerResources/JavaEOInterfaceSwing.jar`
   - `/System/Library/Frameworks/JavaEORuleSystem.framework/WebServerResources/JavaEORuleSystem.jar`
3. Open the project that you want to deploy. Build but don't run the project.
4. Open a terminal window and change directory to the project's `.woa` file (named *ProjectName*.woa). By default, the `.woa` is created in your project's directory.
5. Change directory to Contents -> WebServerResources -> Java.
6. This directory should contain a `.jar` file that contains the application's client-side classes. Add the `.jar` file to the project you created in step 1.
7. In the project you created in step 1, choose Edit Active Target from the Project menu.
8. Under Build Phases, select Frameworks & Libraries. Make sure the checkbox next to Merge is selected for each `.jar` file in the project, as shown in Figure 10-4.

Figure 10-4 Configure merging of `.jar` files



Target "Launcher" of Project "Launcher"

Frameworks & Libraries

Files:

JavaEODistribution.jar	✓ Merge
JavaEOApplication.jar	✓ Merge
JavaEOControl.jar	✓ Merge
JavaEOGeneration.jar	✓ Merge
JavaEOInterface.jar	✓ Merge
JavaEOInterfaceSwing.jar	✓ Merge
JavaEORuleSystem.jar	✓ Merge
JavaFoundation.jar	✓ Merge
classes.jar	✓ Merge

9. Delete all the methods except `main` from the Java class with the same name as the project you created in step 1. So if you named the project “Launcher,” change the file `Launcher.java` to look like Listing 10-1.

Listing 10-1 Launcher class

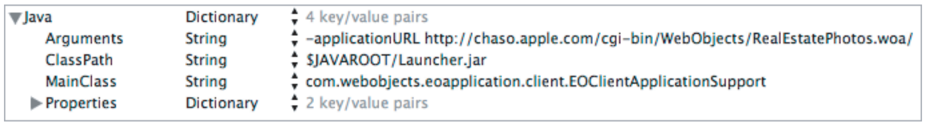
```
public class Launcher extends Object
{

    public static void main(String args[]) {
       com.webobjects.eoapplication.client.EOClientApplicationSupport(new String[]{"",
         ""});
    }

}
```

10. Under Info.plist Entries, select Expert View.

11. Add a key to the Java entry called Arguments, of type String and with a value that is the WebObjects application URL of the server application. The form of the value is `http://`*hostName*`/`*adaptorName*`/WebObjects/`*applicationName*`.woa/`. In development mode, enter the direct connect URL (see the launch console) as shown in Figure 10-5.

12. Modify the `MainClass` key in the Java entry to be `com.webobjects.eoapplication.client.EOClientApplicationSupport`, as shown in Figure 10-5.

Figure 10-5 Add keys to the Info.plist entry



▼Java	Dictionary	4 key/value pairs
Arguments	String	-applicationURL http://chaso.apple.com/cgi-bin/WebObjects/RealEstatePhotos.woa/
ClassPath	String	$JAVAROOT/Launcher.jar
MainClass	String	com.webobjects.eoapplication.client.EOClientApplicationSupport
▶Properties	Dictionary	2 key/value pairs

13. Build the project. The product is a double-clickable Mac OS X application bundle.

Where To Go Next

“Restricting Access to an Application” (page 177) through “Building a Login Window” (page 251) contain information on various tasks you’ll perform when adding features to Java Client applications. Some of the chapters contain scenarios listing specific problems and their solutions. You can use those chapters as a reference guide while building applications.

Table 10-1 provides a high-level overview of the task chapters.

Table 10-1 Task chapter overview

Chapter	Difficulty Level	Task Category
“Restricting Access to an Application” (page 177)	Intermediate	Access control

Chapter	Difficulty Level	Task Category
"Generating Controllers With the Controller Factory" (page 181)	Easy	Programmatic customizations
"Adding Custom Menu Items" (page 185)	Intermediate	XML customizations
"Adding Custom Actions to Controllers" (page 189)	Intermediate	XML, rule, and programmatic customizations
"Common Rules" (page 193)	Easy	Rule customizations
"Freezing XML User Interfaces" (page 199)	Intermediate	XML and rule customizations
"Mixing Static and Dynamic User Interfaces" (page 209)	Advanced	Nib file and rule customizations
"Using Custom Views in Nib Files" (page 213)	Advanced	Nib file customizations
"Using and Extending Image Views in Nib Files" (page 221)	Intermediate	Nib file customizations
"Using Pop-up Menus in Nib Files" (page 227)	Advanced	Nib file customizations
"Localizing Dynamic Components" (page 235)	Advanced	Localization; rule and programmatic customizations
"Building Custom List Controllers" (page 243)	Advanced	Programmatic customizations
"Using HTML on the Client" (page 245)	Advanced	WebObjects HTML integration
"Building a Login Window" (page 251)	Advanced	Access control

Restricting Access to an Application

In a real world application, you'll likely need to restrict access to the application and to functions within the application. In a Java Client application that uses the rule system, you can use rules to accomplish this.

Note: Restricting access to an application's user interface doesn't necessarily restrict access to an application's data. To secure an application's data, you should implement security mechanisms on the distribution layer. See "The Distribution Layer" (page 93) for more information.

The Documents Menu

Problem: The Documents menu in Direct to Java Client applications offers unrestricted access to the entities in the enterprise object models of the application. You want to restrict access to this menu.

Solution: Use the rule system to override the default behavior.

The actions in the Documents menu are defined by the `actions` key in the rule system. You can write a rule overriding `actions` to point to a D2WComponent:

Left-Hand Side:	`*true*`
Key:	`actions`
Value:	`"UserActions"`
Priority:	`50`

See "New D2WComponent" (page 186) to learn how to add a D2WComponent to a project.

The HTML file of the UserActions component is simply an empty array:

```
<ARRAY></ARRAY>
```

If you override `actions` like this, however, your application is unusable since you've locked down all access to it. So you need to provide a custom mechanism to access its functionality. A common mechanism is to use an interface built in Interface Builder. That interface provides buttons or other widgets, which when clicked invoke actions in the application. See "Building the User Interface" (page 251) to learn how to load a nib file when an application starts up. See "Adding Custom Menu Items" (page 185) to learn how to add custom menu items to an application.

The Default Query Window

Problem: When a Direct to Java Client application starts up, the default behavior is to display a query window. From this window users can query on all entities in the application's enterprise object model. You want to change this behavior.

Solution: Use the rule system to override the default behavior.

When an application starts up, the rule system asks for all the available specifications in the application. These specifications are defined in `.d2wmodel` files in the project and in the project's frameworks. Then, the rule system asks for all the default specifications. The default specifications are fired first when the application launches. So, by overriding the default specifications, you control what the user sees when the application launches.

You can override the default specifications with a rule like this:

Left-Hand Side:	`*true*`
Key:	`defaultSpecifications`
Value:	`"BlankSpecifications"`
Priority:	`50`

This rule points the default specifications to a D2WComponent called "BlankSpecifications." The HTML file of the BlankSpecifications component is simply an empty array:

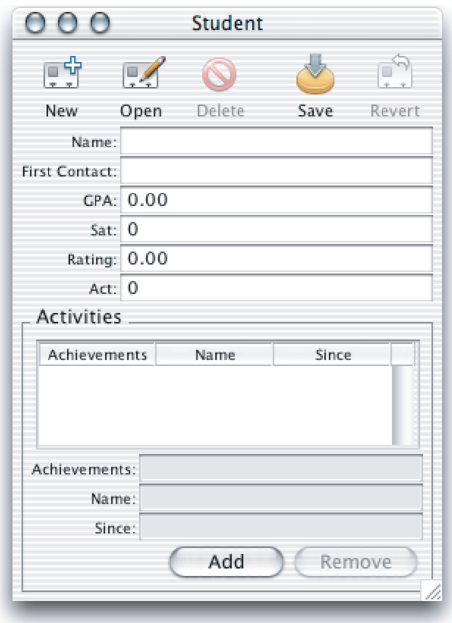
```
<ARRAY></ARRAY>
```

Now, when the application starts up, no windows are displayed on the screen and no menu items appear. So, you have to provide other mechanisms to allow users access to the application's user interface. See "Building a Login Window" (page 251) for some suggestions.

Restricting Tasks Within the Application

Problem: In form windows in Direct to Java Client applications, a number of actions are available to the user by default as shown in Figure 11-1. These actions are insert, open, delete, save, and revert.You want to disable the buttons that invoke some of these actions.

Figure 11-1 Default actions in a form window



Name:	
First Contact:	
GPA:	0.00
Sat:	0
Rating:	0.00
Act:	0
Achievements:	
Name:	
Since:	

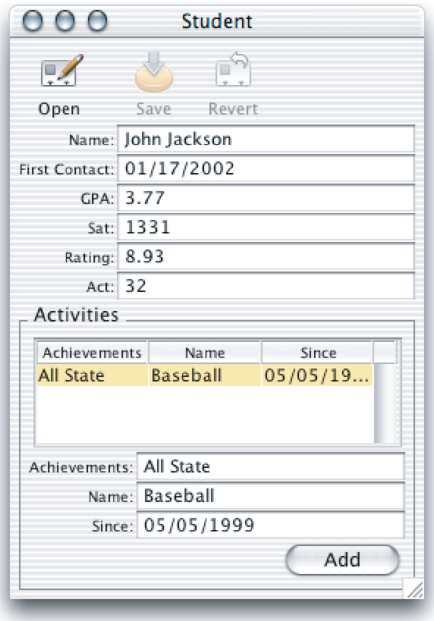
Solution: Use the rule system to override the default behavior.

The rule system provides a key to disable certain actions. By providing the names of the actions you wish to disable as the right-hand side value of this rule, those actions are disabled in all dynamically generated controllers. This and many other rules have no effect on frozen XML or frozen interface files.

Left-Hand Side:	`*true*`
Key:	`disabledActionNames`
Value:	`(insertWithTask, delete)`
Priority:	`50`

This rule disables the insert and delete actions, which is appropriate for the application whose form window is shown in Figure 11-2.

Figure 11-2 Disabled actions in a form window



Name:	John Jackson
First Contact:	01/17/2002
GPA:	3.77
Sat:	1331
Rating:	8.93
Act:	32

Achievements	Name	Since
All State	Baseball	05/05/19...

Achievements:	All State
Name:	Baseball
Since:	05/05/1999

If you're working with frozen XML components, you can remove the `ACTIONSBUTTONCONTROLLER`tags to disable the action buttons in that window. This may be too drastic a measure for your needs, but frozen XML is by definition less flexible than dynamically generated components, and this is one of its costs. If you do remove the `ACTIONSBUTTONCONTROLLER` tags, you can still specify custom action buttons by writing custom controller classes and specifying them with `CONTROLLER` tags in the XML. See "Using a Custom Controller Class in Frozen XML" (page 207) to learn how to write and use custom controller classes.

Generating Controllers With the Controller Factory

Much of the magic behind Direct to Java Client applications happens in the controller factory, the class `com.webobjects.eogeneration.EOControllerFactory`. The purpose of the class is to produce controllers—windows, dialogs, list controllers, select controllers, controllers for particular tasks, and so on. By learning how to use the controller factory programmatically, you can take greater control of Direct to Java Client applications—you can learn to be the magician.

Selecting Objects in an Entity

Problem: You need a user interface and logic to provide a way for users to select an object or objects from a particular table in the data store.

Solution: Use the controller factory to get a select controller for a particular entity.

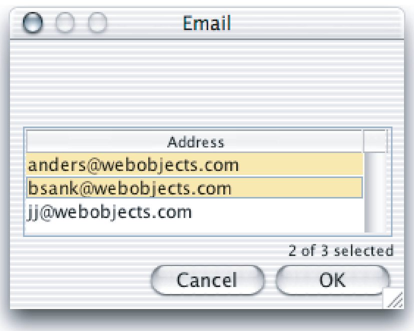
If you tackle this task without using the rule system, you could spend an hour or more in Interface Builder building the user interface and connecting it to a custom controller class to get the selected objects and pass them on to the requesting object. But by using the rule system and the controller factory, a single method invocation does all of this for you.

In a client-side view class (such as the CustomFormController class in "Extend a Controller Class" (page 130) or a subclass of another core controller class), add the import statement for `com.webobjects.eogeneration`. This package contains the controller factory. Then, in the action method that triggers the selection, add this invocation on the controller factory:

```
EOControllerFactory.sharedControllerFactory().selectWithEntityName("entityName",
true, false);
```

The method takes three arguments: the entity to select from, a Boolean value determining whether multiple selections are allowed, and a Boolean value representing whether the insertion of new records is allowed (if the dialog provides an action to add new records). When invoked, the method presents a select dialog like that shown in Figure 12-1.

Figure 12-1 Select dialog



Address
anders@webobjects.com
bsank@webobjects.com
jj@webobjects.com

2 of 3 selected

The method returns an array of EOGlobalID objects representing the selected objects. To get enterprise objects from EOGlobalID objects, you can use the method `objectForGlobalID` defined in `com.webobjects.eocontrol.EOEditingContext`. See the API reference for more information.

Triggering a Task

Problem: You need to provide a custom task to perform some function in the application. You need a way to trigger this task.

Solution: Write a task using a rule and trigger it with an invocation on the controller factory.

Suppose that you have a frozen XML interface in your application. There is no method in the controller factory to simply invoke this frozen interface. But you can easily define a task to do this.

If the frozen XML component is called ImageQueryController, you would define the new task like this:

Left-Hand Side:	`(task ='imageQuery')`
Key:	`window`
Value:	`"ImageQueryController"`
Priority:	`50`

In a client-side view class (not a model class or a controller class), add the import statement for `com.webobjects.eogeneration`. This package contains the controller factory. Then, in the action method that triggers the selection, add this invocation on the controller factory:

```
EOControllerFactory.sharedControllerFactory().openWindowForTaskName("imageQuery");
```

Inserting Objects

Problem: You need to provide a form window for a particular task so a user can insert new records into a table.

Solution: Use the controller factory to get a form controller for a particular entity.

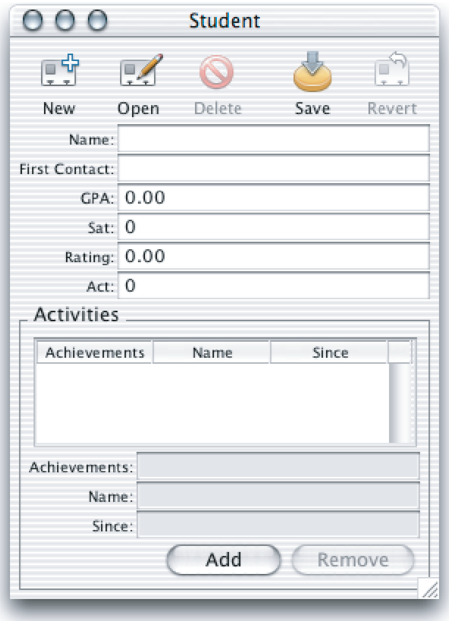
If you implement this feature without using the controller factory or the rule system, you could spend an hour or more in Interface Builder building the interface, connecting the controller, and then writing code to invoke the interface. But by using the rule system and the controller factory, a single method invocation does all of this for you.

In a client-side view class (not a model class or a controller class), add the import statement for `com.webobjects.eogeneration`. This package contains the controller factory. Then, in the action method that triggers the selection, add this invocation on the controller factory:

```
EOControllerFactory.sharedControllerFactory().insertWithEntityName("Document");
```

This method simply takes the name of an entity in the enterprise object model group of your application. It results in a form window like that shown in Figure 12-2.

Figure 12-2 Form window from controller factory



Name:	
First Contact:	
GPA:	0.00
Sat:	0
Rating:	0.00
Act:	0

Achievements	Name	Since

Achievements:	
Name:	
Since:	

Adding Custom Menu Items

There are many ways to add custom menu items to a Direct to Java Client application. This chapter describes the most common mechanisms.Para

About Actions

Before learning how to add actions to your application, you should understand the different kinds of actions in Direct to Java Client applications. If you consult "Controllers and Actions Reference" (page 265), you'll find a number of types of available actions. Whereas the appendix simply lists the XML tags and attributes of each action, this section describes the differences between each type of action and what each attribute represents.

`APPLICATIONACTION`
: These actions are added to the menu specified by the `descriptionPath` parameter. They invoke a method in the application object specified by the `actionName` parameter.

`CONTROLLERHIERARCHYACTION`
: These actions are added to the menu specified by the `descriptionPath` parameter. They invoke a method in the controller hierarchy specified by the `actionName` parameter. This means that the menu items for these types of actions are available only if the action method is defined in the controller hierarchy whose top level controller is active in the application.

`HELPWINDOWACTION`
: These actions are added to the Help menu. They invoke rule system tasks specified by the `task` parameter.

`TOOLWINDOWACTION`
: These actions are added to the Tools menu. They invoke rule system tasks specified by the `task` parameter.

`WINDOWACTION`
: These actions are added to the menu specified by the `descriptionPath` parameter. They invoke rule system tasks specified by the `task` parameter.

The default actions in a Direct to Java Client application are defined in the `com.webobjects.eoapplication` package. As described in "Restricting Access to an Application" (page 177), you can take control of the actions in menus by overriding the `actions` key in the rule system.

You write a rule whose right-hand side key is `actions` and right-hand side value is the name of a D2WComponent in the application that specifies the custom actions.

Note: If you want to use the default actions in a Direct to Java Client application, you specify additional actions with the `additionalActions` key, which also points to a D2WComponent defining the actions.

The following sections describe how to add the different types of actions to your application. Since they all require a D2WComponent, the task for adding one to an application is given first.

New D2WComponent

Problem: You want to add a D2WComponent to your project to hold custom actions (or for other customization purposes).

Solution: Add a WOComponent to your project and make it a D2WComponent.

The first step in adding custom actions is to create a new D2WComponent in which to define them. In a Direct to Java Client project, add a new WebObjects component to the Application Server target. Call it UserActions. This creates a new component of type `com.webobjects.appserver.WOComponent`. However, you need a D2WComponent, so add an import statement for the `com.webobjects.directtoweb` package and change the superclass of UserActions to D2WComponent, as shown in Listing 13-1 (page 186).

Listing 13-1 Changing the superclass of UserActions

```
import com.webobjects.appserver.*;
import com.webobjects.foundation.*;
import com.webobjects.directtoweb.*; // add this

public class UserActions extends D2WComponent { //change superclass to this

    public UserActions(WOContext context) {
        super(context);
    }
}
```

Now you can add actions to this component to provide custom menu items to your application as described in the following sections.

Application-Wide Actions

Problem: You want to add a new menu item that is always available in the client application. The menu item invokes an action method.

Solution: Use `APPLICATIONACTION`.

Suppose your application has a main window that provides access to the application's primary functions. It's conceivable that this window might become hidden underneath other windows as users use the application. So, you can provide a custom menu item that brings this window forward.

You specify the method an `APPLICATIONACTION` object invokes with the `actionName` parameter. The rule system looks for the method in subclasses of the client's principal class, EODynamicApplication (direct project types) or EOApplication (nondirect project types). If the method cannot be found, the menu item is still displayed but it is disabled (grayed out).

In the HTML file of the D2WComponent that contains your application's custom menu items (`UserActions.html` in the UserActions component created with the steps described in "New D2WComponent" (page 186)), the XML description for an `APPLICATIONACTION` object that invokes a method called `bringForwardMainWindow` looks like this:

```
<APPLICATIONACTION actionName="bringForwardMainWindow" menuAccelerator="shift
B" descriptionPath="Window/Main Window"/>
```

This description specifies a custom action that is displayed in the Window menu as the menu item Main Window with the keyboard equivalent Shift-B. It invokes a method called `bringForwardMainWindow` on the client application's principal class. `APPLICATIONACTION` XML descriptions can include other parameters. The possible parameters for XML descriptions of actions are listed in "EOActions XML Descriptions" (page 286).

Note: XML descriptions, when used in the `.html` file of D2WComponents, replace all the HTML in those files.

Menu-Specific Actions

Problem: You want to add a new menu item that is always available in the client application and that invokes a task defined in the rule system. The menu item appears in either the Help menu or the Tools menu.

Solution: Use `HELPWINDOWACTION` or `TOOLWINDOWACTION`.

Suppose your application includes a frozen XML component containing help for the application. The HTML file of the D2WComponent containing your custom rules would include this XML description:

```
<HELPWINDOWACTION task="help" menuAccelerator="shift T"
descriptionPath="Window/Main Window"/>
```

If the frozen XML component containing the help file is named HelpWindow, the rule to load it is as follows:

Left-Hand Side:	`(task='help')`
Key:	`window`
Value:	`"HelpWindow"`
Priority:	`50`

This defines a new task that opens the frozen XML component specified in the right-hand side value.

To add a menu item to the Tools menu, follow the steps for adding an item to the Help menu, changing `HELPWINDOWACTION` to `TOOLWINDOWACTION` in the XML description.

Controller-Specific Actions

Problem: You want to add a new menu item that is available only in the client application for a particular controller hierarchy. The menu item invokes an action in a particular controller hierarchy.

Solution: Use a `CONTROLLERHIERARCHYACTION`.

Sometimes you want a menu item to be available only while a particular task or user interface component is active. For example, in "Enhancing the Application" (page 103), a custom action is added to the application to send a report of a student's information. In the tutorial, the custom action is added only to form windows for the Student entity, but this would also be a good action to add as a menu item.

However, this menu item should be available only if a student record is in the frontmost window. So `CONTROLLERHIERARCHYACTION` is the appropriate type of action. These actions are enabled only if the action method is in a class that is part of the controller hierarchy represented in the frontmost window of an application.The HTML file of the D2WComponent containing your custom rules would include this XML description:

```
<CONTROLLERHIERARCHYACTION actionName="activateMainWindow" menuAccelerator="shift
 A" descriptionPath="Window/Main Window"/>
```

This invokes a method called `activateMainWindow` in a class that is part of the frontmost controller hierarchy.

Adding Custom Actions to Controllers

When building Direct to Java Client applications, it's common to want to add actions to the application's controllers. The default actions in a form controller are insert, delete, revert, save, and open. There are many ways to add actions to controllers and still preserve the dynamism of the application. This chapter describes some of the possible ways.

Subclassing Controller Classes

Problem: You need to add actions to a controller yet still preserve the dynamic character of the controller.

Solution: Subclass the controller class and use the rule system to use it throughout the application.

This technique is used in "Extend a Controller Class" (page 130) in the chapter "Enhancing the Application" (page 103).

Subclassing a controller class and writing a rule to use it is the best way to add custom actions to your application's controllers. As well as taking real advantage of object-oriented programming, it preserves the dynamism of Direct to Java Client applications. The other mechanisms to add actions require freezing XML, and anytime you freeze XML, you lose a lot of the dynamism of the rule system.

The dynamically generated user interfaces in Java Client rely on a core set of controller classes: EOFormController, EOQueryController, and EOListController. In an application that, for example, stores images in records, you need custom actions to both select images from the file system and download them to the file system. This requires two additional action buttons in a form window, Download Image and Update Image.

To add these actions, create a new class called FormController, as shown in Listing 14-1.

Listing 14-1 Subclassing EOFormController

```
package assetmanager.client;

import javax.swing.*;
import com.webobjects.foundation.*;
import com.webobjects.eocontrol.*;
import com.webobjects.eoapplication.*;
import com.webobjects.eogeneration.*;
import com.webobjects.eodistribution.client.*;

public class FormController extends EOFormController {

    public FormController(EOXMLUnarchiver unarchiver) {
        super(unarchiver);
    }

    public NSArray defaultActions() {
        Icon icon =
            EOUserInterfaceParameters.localizedIcon("ActionIconInspect");
        NSMutableArray actions = new NSMutableArray();
        actions.addObject(EOAction.actionForControllerHierarchy("saveImageToDisk",
           "Download Image", "Download Image", icon, null, null, 300, 50, false));

        icon = EOUserInterfaceParameters.localizedIcon("ActionIconOk");
        actions.addObject(EOAction.actionForControllerHierarchy("updateImageInRecord",
            "Update Image", "Update Image", icon, null, null, 300, 50, false));
        return EOAction.mergedActions(actions, super.defaultActions());
    }

    public boolean canPerformActionNamed(String actionName) {
        return actionName.equals("saveImageToDisk") ||
           super.canPerformActionNamed(actionName));
    }

    public void saveImageToDisk() {
        //some code
    }

    public void updateImageInRecord() {
        //some code
    }
}
```

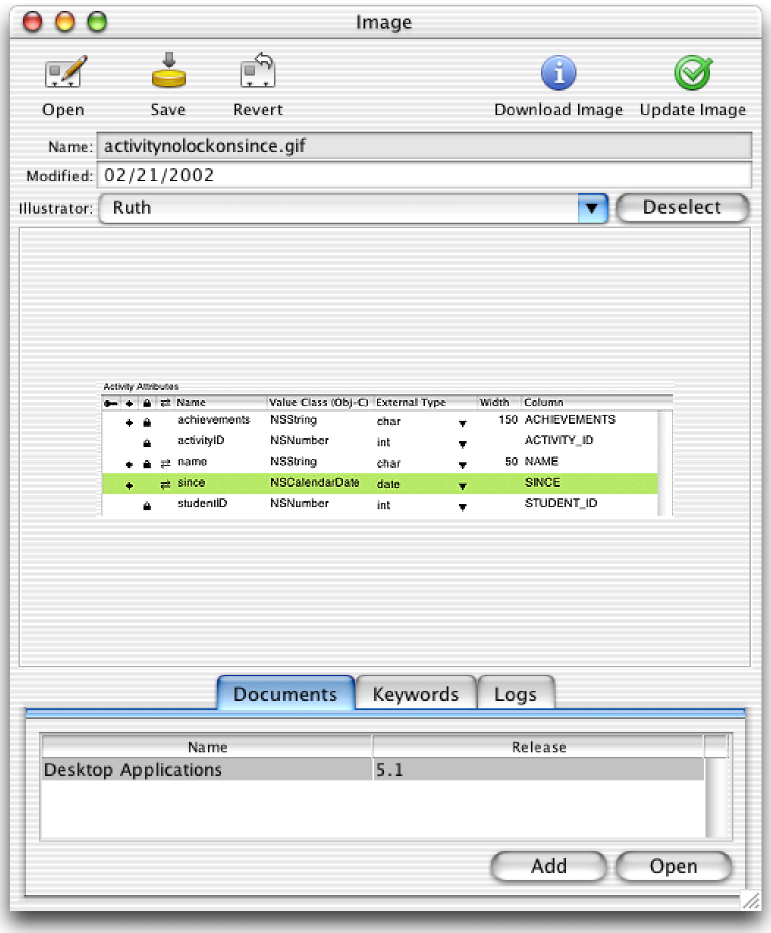
Subclasses of the core controller classes must contain these methods: a method overriding `defaultActions`, a method overriding `canPerformActionNamed`, and a method for each action defined in `defaultActions`. By overriding `defaultActions`, you are adding to the controller's actions, and by overriding `canPerformActionNamed`, you are authorizing the additional actions.

To use this class in all form windows throughout the application, you need only write a simple rule:

Left-Hand Side:	`((task='form') and (controllerType='entityController'))`
Key:	`className`
Value:	`"assetmanager.client.FormController"`
Priority:	`50`

So, without needing to freeze XML, these customizations change the default form window to include new actions, as shown in Figure 14-1.

Figure 14-1 Image form window with new actions



Name: activitynolockonsince.gif
Modified: 02/21/2002
Illustrator: Ruth

Activity Attributes

				Name	Value Class (Obj-C)	External Type	Width	Column
	◆	🔒		achievements	NSString	char	150	ACHIEVEMENTS
		🔒		activityID	NSNumber	int		ACTIVITY_ID
	◆	🔒	⇄	name	NSString	char	50	NAME
	◆		⇄	since	NSCalendarDate	date		SINCE
		🔒		studentID	NSNumber	int		STUDENT_ID

Name	Release
Desktop Applications	5.1

The standard actions delete and insert are disabled by another rule:

Left-Hand Side:	`*true*`
Key:	`disabledActionNames`
Value:	`(insertWithTask, delete)`
Priority:	`50`

This rule is described in "Restricting Access to an Application" (page 177).

Writing Custom Controller Classes

Problem: For any number of reasons, subclassing the core controller classes to provide custom actions doesn't meet your needs.

Solution: Subclass EOController and write a rule or XML to use it.

This mechanism of writing a custom action is very similar to that described in "Subclassing Controller Classes" (page 189), except that you subclass EOController instead of an entity-level controller like EOFormController and EOQueryController. Listing 14-2 (page 192) shows the code for the class that adds an action that displays a simple information dialog.

Listing 14-2 A custom controller class

```
package businesslogic.client;
import java.awt.event.*;
import javax.swing.*;
import com.webobjects.foundation.*;
import com.webobjects.eoapplication.*;
import com.webobjects.eogeneration.*;

public class NewController extends EOController {

    public NewController(EOXMLUnarchiver unarchiver) {
        super(unarchiver);
    }

    protected NSArray defaultActions() {
        Icon icon =
           EOUserInterfaceParameters.localizedIcon("ActionIconInspect");
          NSMutableArray actions = new NSMutableArray();
            actions.addObject(EOAction.actionForControllerHierarchy("runInfoDialog",
"Run
            Info Dialog", "Run Info Dialog", icon, null, null, 300, 50, false));

        return EOAction.mergedActions(actions, super.defaultActions());
    }

    public boolean canPerformActionNamed(String actionName) {
        return actionName.equals("runInfoDialog") ||
                super.canPerformActionNamed(actionName);
    }

    public void runInfoDialog() {
        EODialogs.runInformationDialog("Hello World!", "Hello World!");
    }
```

The most common way to use this custom controller in an application is in a frozen XML component. You can add a `CONTROLLER` tag specifying the fully qualified class name of the new class:

```
CONTROLLER className="businesslogic.client.NewController"/>
```

You can also write a rule to use the custom controller:

Left-Hand Side:	`((task='query') and (controllerType='entityController'))`
Key:	`className`
Value:	`"businesslogic.client.NewController"`
Priority:	`50`

Common Rules

This chapter provides examples of some common rules you can use to customize applications. It first provides a section listing common left-hand side qualifier keys. You use these keys to tell the rule system which part of an application to affect with a particular rule.

Left-Hand Side Qualifier Keys

Most rules that you write are likely to use the following keys in the left-hand side qualifier:

- `task`: A name of the task for which information is requested from the rule system. The values used in the default rule system are
  - `form` (editing an enterprise object)
  - `list` (listing multiple enterprise objects)
  - `query` (qualifying enterprise objects for a search)
  - `identify` (describing enterprise objects in a short form)
  - `select` (a combination of query and list)
  - `editor` (a combination of list and form)
  - `queryWindow` (the task for requesting the complete query window)
  - `enumerationWindow` (the task for requesting the complete enumeration window)
- `entity`: The entity the rule system works on for a particular rule. You most often identify an entity using the key path `entity.name`.
- `propertyKey`; `attribute`; `relationship`: The property key the rule system works on for a particular rule. You can use `attribute.name` or `relationship.name` rather than `propertyKey`. You'd likely do this if an entity contains ambiguously named property keys, such as an attribute and relationship that have the same name.

You can further restrict a left-hand side qualifier using these keys:

- `question`: A question the rule system is supposed to answer. The most important question keys in the rule system are
  - `window`
  - `modalDialog`
  - `controller`
  - `propertyKeys`
  - `actions`

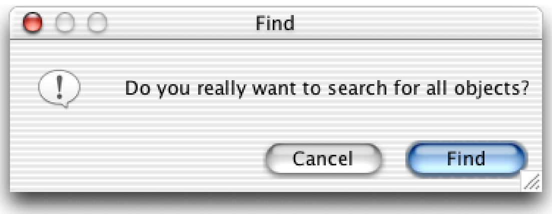
- ❏ `availableSpecifications`
- ❏ `defaultSpecifications`
- ❏ `userInterfaceParameters`

- `controllerType`: An indication of what kind of controller the rule system works on for a particular rule. The values in the default rule system are
  - ❏ `actionWidgetController`
  - ❏ `dividingController`
  - ❏ `groupingController`
  - ❏ `entityController`
  - ❏ `inspectorController`
  - ❏ `modalDialogController`
  - ❏ `statisticsController`
  - ❏ `tableController`
  - ❏ `splitController`
  - ❏ `widgetController`
  - ❏ `windowController`
- `isRootController`: Specifies whether the rule system is working on the root controller (in which case this key evaluates to `null`) or a subcontroller (in which case this key evaluates to `false`).
- `rootQuestion`, `rootTask`, `rootEntity`: These represent copies of the `question`, `task`, and `entity` values of the original request triggered by the controller factory. For example, through `rootTask` you can find out whether you are in a `form` window.

Confirmation Dialog

Problem: By default when you query on an entity without supplying a qualifier, you are presented with a dialog to confirm the action, as shown in Figure 15-1. This behavior is intended to warn users about performing unqualified queries of the data store, which could fetch hundreds or thousands of records.

Figure 15-1 Confirm dialog on unqualified queries



Do you really want to search for all objects?

Solution: Use the rule system to override the default behavior.

The confirmation dialog appears when a user invokes an unqualified query (a query with no search criteria). Whenever you want to modify the behavior of a controller in a Direct to Java Client application, you should first consult "Controllers and Actions Reference" (page 265). If you look for EOQueryController, you'll find an XML attribute called `runsConfirmDialogForEmptyQualifiers`. This is the switch you're looking for. So to disable the confirmation dialog, add this rule to your application's `d2w.d2wmodel` file:

Left-Hand Side:	`*true*`
Key:	`runsConfirmDialogForEmptyQualifiers`
Value:	`"false"`
Priority:	`50`

Window Size

Problem: You want to specify a minimum width and height for all windows in your application.

Solution: Use the rule system.

To set the minimum width of all windows in your application to 512 pixels, use this rule:

Left-Hand Side:	`(controllerType='windowController')`
Key:	`minimumWidth`
Value:	`512`
Priority:	`50`

To set the minimum height of all windows in your application to 350 pixels, use this rule:

Left-Hand Side:	`(controllerType='windowController')`
Key:	`minimumHeight`
Value:	`350`
Priority:	`50`

Widget Alignment

Problem: You want to right-align all widgets that contain numbers.

Solution: Write a rule.

To right-align all widgets in your application whose attribute's value class is a number type, use this rule:

Left-Hand Side:	`((not (attribute= nil)) and (attribute.valueClassName='NSNumber') or (attribute.valueClassName='NSDecimalNumber'))`
Key:	`alignment`
Value:	`"Right"`

Priority:	`50`

Custom Controllers

Problem: You've subclassed one of the core controller classes (EOFormController, EOListController, EOQueryController) and you want to use it in place of the default controller throughout the application.

Solution: Write a rule.

To use a custom subclass of EOQueryController called QueryController in the package `com.mypackage`, use this rule:

Left-Hand Side:	`((task='query')and (controllerType='entityController'))`
Key:	`className`
Value:	`"com.mypackage.QueryController"`
Priority:	`50`

To use the custom subclass only for a specific entity, use this rule:

Left-Hand Side:	`((task='query') and (entity.name="<entity name>"))`
Key:	`className`
Value:	`"com.package.CustomQueryController"`
Priority:	`50`

Custom Class for Widgets

Problem: You want to use a custom widget class for a particular widget in your application.

Solution: Write a rule.

To use a custom widget class for the `creditCardNumber` attribute of a Person entity, use this rule:

Left-Hand Side:	`((entity.name='Person') and (attribute.name="creditCardNumber"))`
Key:	`widgetController`
Value:	`"com.client.CustomController"`
Priority:	`50`

CustomController is a subclass of EOWidgetController:

```
package com.client;

import javax.swing.*;
import com.webobjects.foundation.*;
import com.webobjects.eogeneration.*;
import com.webobjects.eointerface.swing.*;
```

```
public class CustomController extends EOWidgetController {

    public CustomController(EOXMLUnarchiver unarchiver) {
        super(unarchiver);
    }

    protected JComponent newWidget() {
        return new JPasswordField("");
    }
}
```

Custom Attributes for Controllers

Problem: You want to set custom attributes for a particular type of controller throughout your application.

Solution: Write a rule.

The Direct to Java Client Assistant allows you to set attributes for controllers such as `horizontallyResizable`, `editability`, and `label`. The attributes for each controller are listed in "Controllers and Actions Reference" (page 265). To disable horizontal resizing for all modal dialogs in an application, use this rule:

Left-Hand Side:	`(controllerType = "modalDialogController")`
Key:	`horizontallyResizable`
Value:	`"false"`
Priority:	`50`

Continuous Change Notification

Problem: You want to implement continuous change notification for a particular controller.

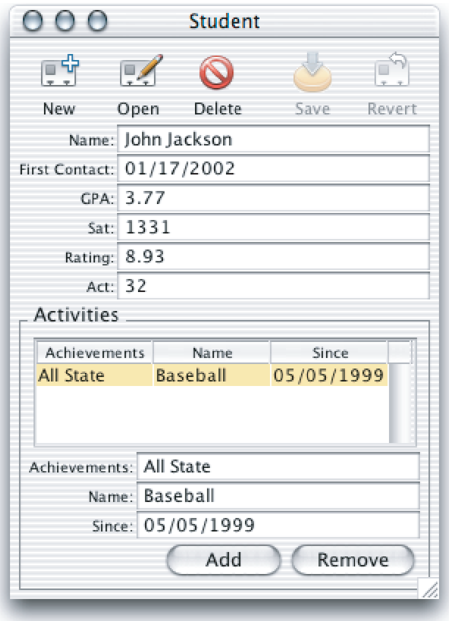
Solution: Write a rule.

Starting with WebObjects 5.2, some controllers can implement continuous change notification. To implement continuous change notification for a dynamically generated controller using rules, you must first identify the controller and then write a rule with a right-hand side key of `prefersContinuousChangeNotification` and value of `true`.

The example application Admissions in "Enhancing the Application" (page 103) could benefit from continuous change notification. In the section "Add Custom Code" (page 114), you added custom business logic that calculates a rating for a particular student record. By default, the rating is updated after the user tabs out of one of the fields that is part of the rating calculation. By applying continuous change notification to one of those fields, however, the rating updates each time a user makes a change in one of those fields.

The user interface element to which you'll add continuous change notification is the Student form window, as shown in Figure 15-2.

Figure 15-2 Field to add continuous change notification to



Name:	John Jackson
First Contact:	01/17/2002
GPA:	3.77
Sat:	1331
Rating:	8.93
Act:	32

Achievements	Name	Since
All State	Baseball	05/05/1999

Achievements:	All State
Name:	Baseball
Since:	05/05/1999

The first step is to identify the form controller in the rule system. The best way to do this is to open Assistant and make a trivial change to that controller, such as changing the default window location. Doing so writes a rule to the application's `user.d2wmodel` file. That rule's left-hand side is how the controller is identified in the rule system. By copying the left-hand side of that rule to the application's `d2w.d2wmodel` file, you can customize its behavior and characteristics by adding right-hand side keys and values.

Here is the rule to implement continuous change notification in this example:

Left-Hand Side:	`((task = 'form') and (entity.name = 'Student'))`
Key:	`prefersContinuousChangeNotification`
Value:	`"true"`
Priority:	`50`

If you implemented the validation methods in "Validation" (page 116), you may see many validation exception dialogs when you implement continuous change notification, depending on what kind of validation you implement.

Freezing XML User Interfaces

You can use the tutorial project you created earlier in "Enhancing the Application" (page 103) as the basis for the exercises in this chapter.

Freeze XML User Interfaces

Problem: You need more finely grained control over the user interface than the Direct to Java Client Assistant allows.

Solution: Use the XML generated by Assistant as a starting point, then edit it by hand to suit your needs.

Freezing XML is another way to customize Direct to Java Client applications. While Assistant allows you to make basic user interface customizations to your application, it is necessarily limited. Freezing XML, however, gives you finer control over your application's user interface. With that said, you should use Assistant as much as you can since freezing XML makes your application more complex and less flexible than just using Assistant.

Freezing XML involves these steps:

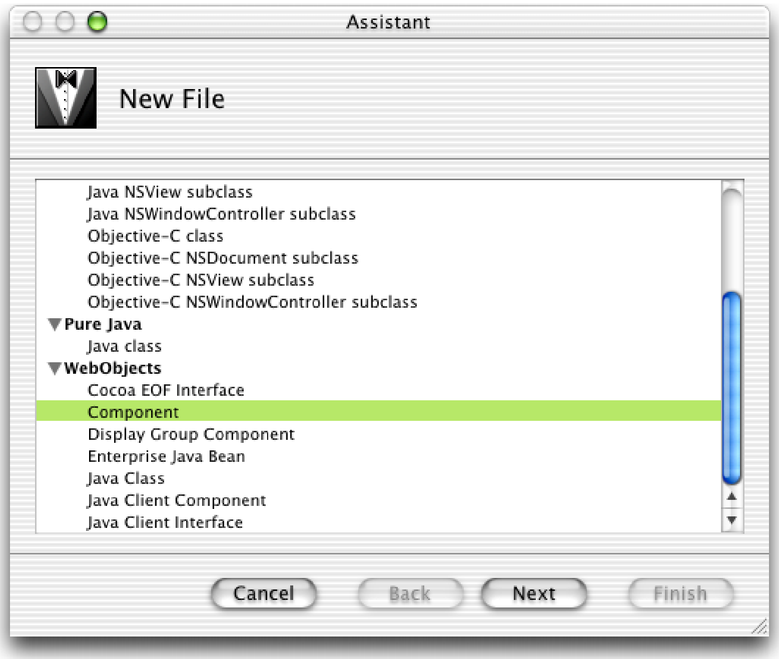
- making a new D2WComponent
- copying an XML description from Assistant and editing it in the new component's `.html` file
- writing a rule to tell the rule system to use the XML component

Note: Before freezing XML, your data model should be as complete as possible. When just using Assistant to customize applications, changes to data models are automatically picked up in most cases. However, the more advanced customization techniques, starting with XML freezing, make data model–user interface synchronization more difficult.

Follow these steps to customize the user interface of the Admissions application using frozen XML:

1. In Project Builder, select the Web Components group, choose File > New File, and select Component from the WebObjects list, as shown in Figure 16-1. Do not select Display Group Component or Java Client Component.

Figure 16-1 Select Component as the file type

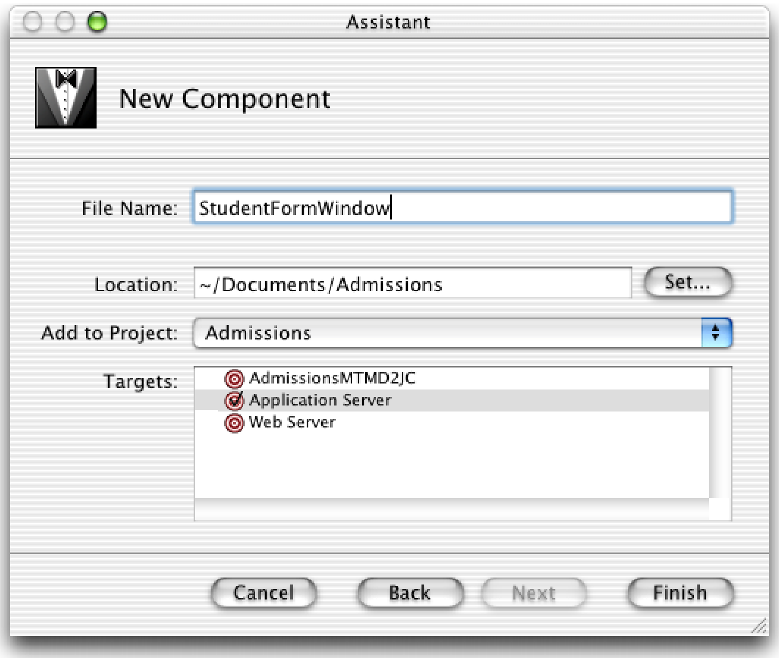


2. Name the new component `StudentFormWindow` and make sure Application Server is the selected target, as shown in Figure 16-2.

   The recommend convention for naming frozen XML components is *EntityNameTaskNameWindowType*. So, if the entity in question is Student, and the task is query, the frozen XML component should be named `StudentQueryWindow`.

Click Finish.

Figure 16-2 Name new component "StudentFormWindow"



File Name: StudentFormWindow

Location: ~/Documents/Admissions

Add to Project: Admissions

Targets: AdmissionsMTMD2JC, Application Server, Web Server

3. The Project Builder assistant for new component files creates a standard WebObjects component, so you need to change it to a D2WComponent. Add the import statement for `com.webobjects.directtoweb` and change the superclass of `StudentFormWindow` to `D2WComponent`, as shown in Listing 16-1 (page 201).

Listing 16-1 Change the superclass of StudentFormWindow to D2WComponent

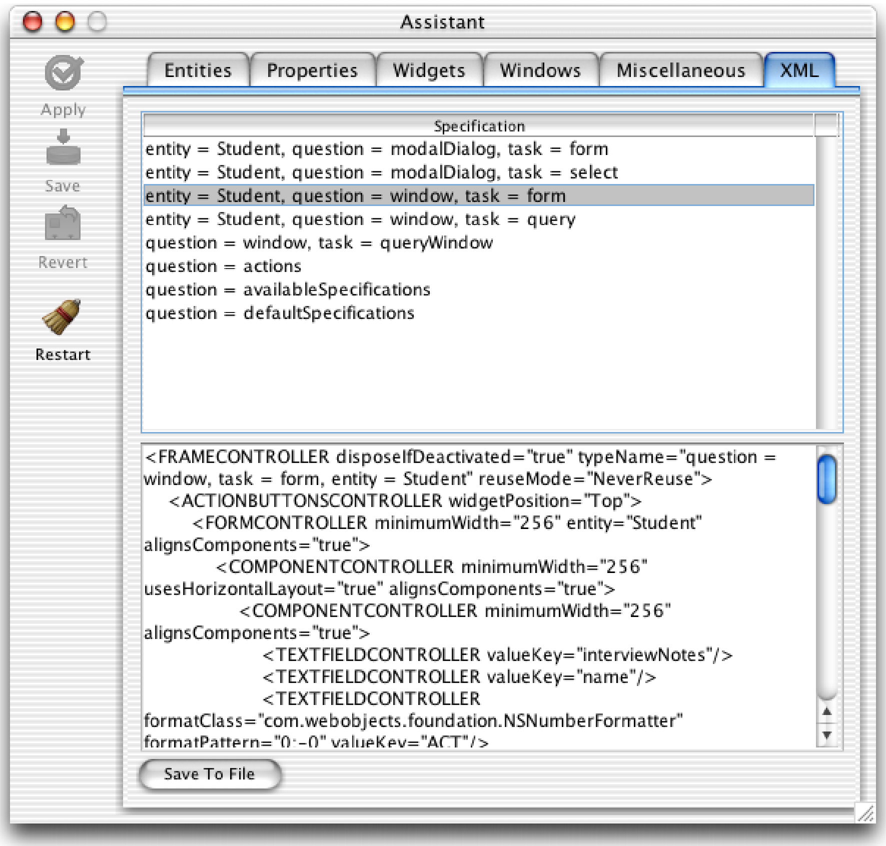
```
import com.webobjects.foundation.*;
import com.webobjects.appserver.*;
import com.webobjects.eocontrol.*;
import com.webobjects.eoaccess.*;
import com.webobjects.directtoweb.*;

public class StudentFormWindow extends D2WComponent {
    public StudentFormWindow(WOContext context) {
        super(context);
    }
}
```

4. Build and run the application and start the client application.

5. Switch to the XML pane in Assistant.

6. Under Specification, choose entity > Student, question > window, and task > form. This puts the XML description for that selection in the XML window as shown in Figure 16-3.

Figure 16-3 XML description of Student entity, form window



Specification
entity = Student, question = modalDialog, task = form
entity = Student, question = modalDialog, task = select
entity = Student, question = window, task = form
entity = Student, question = window, task = query
question = window, task = queryWindow
question = actions
question = availableSpecifications
question = defaultSpecifications

```
<FRAMECONTROLLER disposeIfDeactivated="true" typeName="question =
window, task = form, entity = Student" reuseMode="NeverReuse">
    <ACTIONBUTTONSCONTROLLER widgetPosition="Top">
        <FORMCONTROLLER minimumWidth="256" entity="Student"
alignsComponents="true">
            <COMPONENTCONTROLLER minimumWidth="256"
usesHorizontalLayout="true" alignsComponents="true">
                <COMPONENTCONTROLLER minimumWidth="256"
alignsComponents="true">
                    <TEXTFIELDCONTROLLER valueKey="interviewNotes"/>
                    <TEXTFIELDCONTROLLER valueKey="name"/>
                    <TEXTFIELDCONTROLLER
formatClass="com.webobjects.foundation.NSNumberFormatter"
formatPattern="0;-0" valueKey="ACT"/>
```

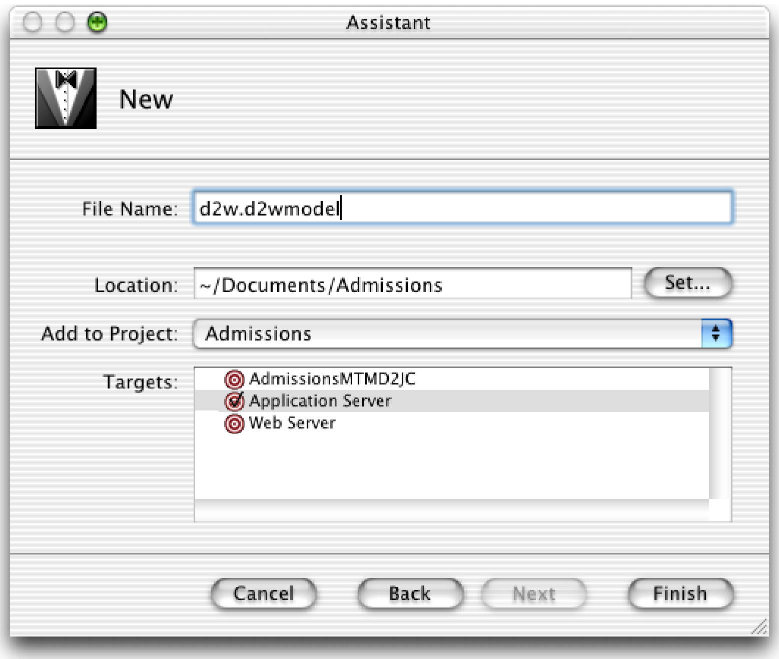
7. Copy the whole XML specification and paste it into the `StudentFormWindow.html` file in Project Builder. `StudentFormWindow.html` is in the StudentFormWindow component.

8. In Project Builder, select the Resources group and choose File > New File and select Empty File. Name the file `d2w.d2wmodel` and make sure Application Server is the selected target, as shown in Figure 16-4 (page 203). This file will hold custom rules you write. Skip this step if your project already has a `d2w.d2wmodel` file.

   You need to have a `d2w.d2wmodel` file for a few reasons. First, Assistant stores its rules in the `user.d2wmodel` file and writes out this file whenever it saves. So, any rules you add or change manually in the `user.d2wmodel` file is wiped out by Assistant.

By writing rules in a separate file, you can maintain a custom set of rules and still use Assistant for basic customizations. At runtime, all the `user.d2wmodel` files in the frameworks and all the `d2w.d2wmodel` files in your project and in your project's frameworks are merged, so the rule system picks up your custom rules and the rules you specified with Assistant, along with all the default rules.

Figure 16-4 Make a new rule file for custom rules



File Name: d2w.d2wmodel

Location: ~/Documents/Admissions

Add to Project: Admissions

Targets: AdmissionsMTMD2JC, Application Server, Web Server

9. Put the Rule Editor application (found in `/Developer/Applications/`) in the Dock. Drag the `d2w.d2wmodel` file to the Rule Editor icon in the Dock to open it.

10. Click New to make a new rule and add these arguments to the left-hand side: `(task = 'form') and (entity.name = 'Student')`.

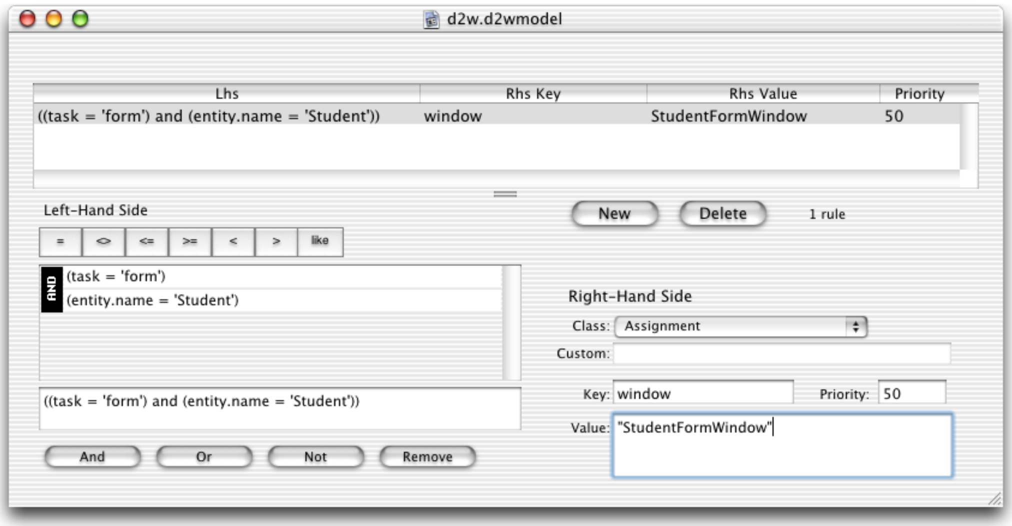
    Collectively, the left-hand side arguments constitute the rule's condition. If the condition exists (that is, the user or application performs some action that triggers the condition), the rule fires and the right-hand side of the rule is evaluated.

    In this case, if the form task is triggered (usually by a user action) on the Student entity, the condition of this rule is `true`, so the rule fires. Collectively, the left-hand side arguments ask "how should this part of the application behave?" And since the condition has been triggered, the behavior of this part of the application is changed per the right-hand side arguments.

    As described in "Task Pop-Up Menu" (page 87), Direct to Java Client applications have four basic tasks: query, form, list, and identify. (The rule system defines other tasks with which you usually do not need to interact). In this step, specifying `task = 'form'` tells the rule system that this rule pertains to the form task. By specifying the entity with `entity.name = 'Student'`, the rule system knows that this rule pertains to the form task for the Student entity. However, if you want to use the frozen XML window for the query task, you would instead specify `task = 'query'`.

11. Set the right-hand side key to `window` and the value to `StudentFormWindow`. Set the priority to 50. Refer to Figure 16-5 for clarity. For an explanation of rule system priorities, see "Rule System Priorities" (page 164).

Figure 16-5 Add a rule to use frozen XML



Lhs	Rhs Key	Rhs Value	Priority
((task = 'form') and (entity.name = 'Student'))	window	StudentFormWindow	50

(task = 'form')
(entity.name = 'Student')

((task = 'form') and (entity.name = 'Student'))

Class: Assignment
Custom:
Key: window
Priority: 50
Value: "StudentFormWindow"

The right-hand side arguments constitute the answer to the question posed in the left-hand side arguments. The answer is made up of a key and a value for that key. In this case, the key is `window` and the value is `StudentFormWindow`. So in this case, the answer is "use the StudentFormWindow as the window for form tasks for the Student entity."

For high-level questions like `controller`, `window`, and `modalDialog`, the rule system expects the value to be the name of a D2WComponent, like StudentFormWindow or any of the default D2WComponent classes defined in `com.webobjects.eogeneration.*;` (see `/System/Library/Frameworks/JavaEOGeneration.framework/Resources/`).

12. Save the `.d2wmodel` file.

Customize the XML

Now that you've successfully frozen XML, you need to customize it to see any benefit. The default Student form window generated by the EOGeneration framework isn't too bad, but you might want to group Student's attributes in a box controller for a cleaner look. Assistant doesn't give you this level of control of the user interface, so you need to edit the XML by hand.

If you closely examine the XML, you'll notice that the widgets are organized in a hierarchy of controllers. The window is defined by a `FRAMECONTROLLER` tag, the action buttons by an `ACTIONSBUTTONCONTROLLER` tag, the form elements by a `FORMCONTROLLER` tag, and the components of the form by `COMPONENTCONTROLLER` tags. You'll notice that the `COMPONENTCONTROLLER` tag for the form that contains the attributes of the Student entity includes two nested `COMPONENTCONTROLLER` tags. You can group Student's attributes into a box by adding a `BOXCONTROLLER` tag between Student's outermost `COMPONENTCONTROLLER` tag and its first inner `COMPONENTCONTROLLER` tag. Add a `BOXCONTROLLER` tag with the following XML (also see code line 1 in Listing 16-2 (page 205)):

```
<BOXCONTROLLER usesTitleBorder="false" highlight="true"
 border="RaisedBezel">
```

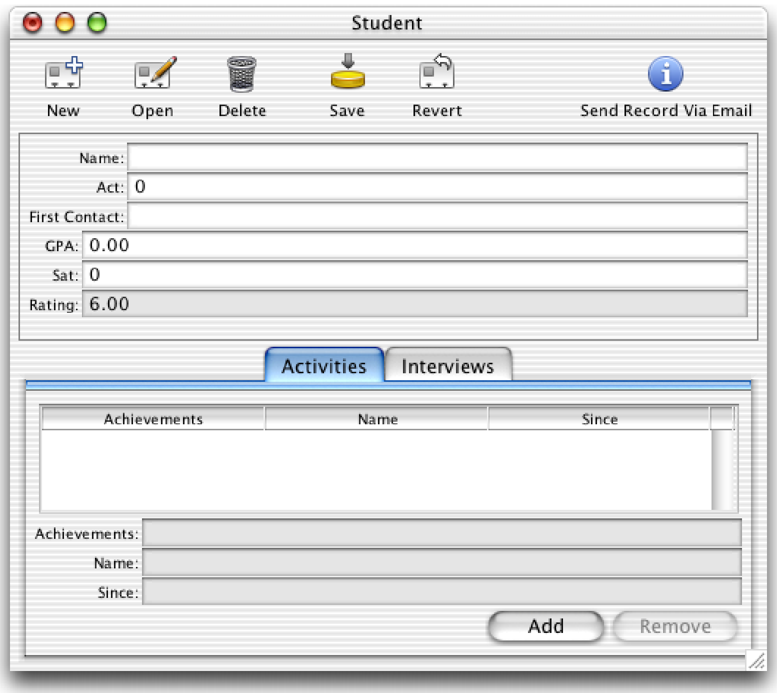
The beginning of the `StudentFormWindow.html` file should look like Listing 16-2 (page 205). Make sure to also add a closing tag for the box controller `</BOXCONTROLLER>` before the closing tag of Student's outermost `COMPONENTCONTROLLER` tag, as shown in code line 2 in Listing 16-2 (page 205).

Listing 16-2 `StudentFormWindow.html` (frozen XML)

```
<FRAMECONTROLLER disposeIfDeactivated="true" typeName="question = window, task = form,
   entity = Student" reuseMode="NeverReuse">
    <ACTIONBUTTONSCONTROLLER widgetPosition="Top">
        <FORMCONTROLLER className="admissions.client.CustomFormController"
            alignsComponents="true" entity="Student" minimumWidth="256">
            <COMPONENTCONTROLLER minimumWidth="256" usesHorizontalLayout="true"
                alignsComponents="true">
                <BOXCONTROLLER usesTitleBorder="false" highlight="true"           // 1
                    border="RaisedBezel">                                          // 2
                <COMPONENTCONTROLLER minimumWidth="256" alignsComponents="true">
                    <TEXTFIELDCONTROLLER valueKey="name"/>
                    <TEXTFIELDCONTROLLER
                        formatClass="com.webobjects.foundation.NSNumberFormatter"
                        formatPattern="0;-0" valueKey="act"/>
                    <TEXTFIELDCONTROLLER
                        formatClass="com.webobjects.foundation.NSTimestampFormatter"
                        formatPattern="MM/dd/yyyy" valueKey="firstContact"/>
                </COMPONENTCONTROLLER>
                <COMPONENTCONTROLLER minimumWidth="256" alignsComponents="true">
                    <TEXTFIELDCONTROLLER
                        formatClass="com.webobjects.foundation.NSNumberFormatter"
                       label="GPA" formatPattern="#,##0.00;-#,##0.00" valueKey="gpa"/>
                    <TEXTFIELDCONTROLLER
                        formatClass="com.webobjects.foundation.NSNumberFormatter"
                   formatPattern="0;-0" valueKey="sat"/>
                    <TEXTFIELDCONTROLLER editability="Never"
                        formatClass="com.webobjects.foundation.NSNumberFormatter"
                        formatPattern="#,##0.00" valueKey="rating"/>
                </COMPONENTCONTROLLER>
                </BOXCONTROLLER>                                                   // 3
```

Figure 16-6 shows an example of a Student form window with the new `BOXCONTROLLER`.

Figure 16-6 Student form window with `BOXCONTROLLER` tag



Name:	
Act:	0
First Contact:	
GPA:	0.00
Sat:	0
Rating:	6.00
Achievements:	
Name:	
Since:	

Note: You can also use some of the WebObjects dynamic elements in frozen XML components such as WOConditional and WORepetition. However, you can only use these elements to control the XML that is generated. You cannot use them to display HTML (only data that is resolved from the controllers is displayed).

Adding Actions to Frozen XML

Problem: You need to add custom actions to a frozen XML component.

Solution: Specify an action method in a business logic class or write a custom controller class.

Edit XML by Hand

To add an action to a frozen XML component, you embed an `ACTIONCONTROLLER` tag inside an `ACTIONBUTTONSCONTROLLER` block (or elsewhere, depending on where you want the button) in a frozen XML file:

```
<ACTIONCONTROLLER label="Send Record Via Email" usesButton="false"
usesAction="true" iconName="ActionIconOk" actionKey="sendRecordViaEmail">
</ACTIONCONTROLLER>
```

Implement the custom action method in the client-side business logic class.

Using a Custom Controller Class in Frozen XML

You can also add actions to frozen XML components by using a custom controller class.

To do this, create an empty Java class file (File > New File, then select “Java class” in the Pure Java group) in Project Builder. Name the new file `NewController` and add it to the Web Server target. Add the import statements and methods shown in the code listing here:

```
package businesslogic.client;
import java.awt.event.*;
import javax.swing.*;
import com.webobjects.foundation.*;
import com.webobjects.eoapplication.*;
import com.webobjects.eogeneration.*;

public class NewController extends EOController {

public NewController(EOXMLUnarchiver unarchiver) {
        super(unarchiver);
    }

    protected NSArray defaultActions() {
        Icon icon =
         EOUserInterfaceParameters.localizedIcon("ActionIconInspect");
        NSMutableArray actions = new NSMutableArray();
       actions.addObject(EOAction.actionForControllerHierarchy("runInfoDialog",
 "Run Info Dialog", "Run Info Dialog", icon, null, null, 300, 50, false));
        return EOAction.mergedActions(actions, super.defaultActions());
    }

    public boolean canPerformActionNamed(String actionName) {
        return actionName.equals("sendRecordViaEmail") ||
                 super.canPerformActionNamed(actionName);
    }

    public void sendRecordViaEmail() {
        EODialogs.runInformationDialog("Hello World!", "Hello World!");
    }
}
```

By overriding `defaultActions`, you are adding to the actions that are displayed in the user interface by the `ACTIONBUTTONSCONTROLLER` tags. See the API reference for `EOApplication.defaultActions` for a description of the parameters.

Notice in the `defaultActions` method that a custom icon is specified using `EOUserInterfaceParameters.localizedIcon`. The method takes a string that is the name of an icon in the Web Server target. You should group all resources such as images in the Web Resources group in your project.

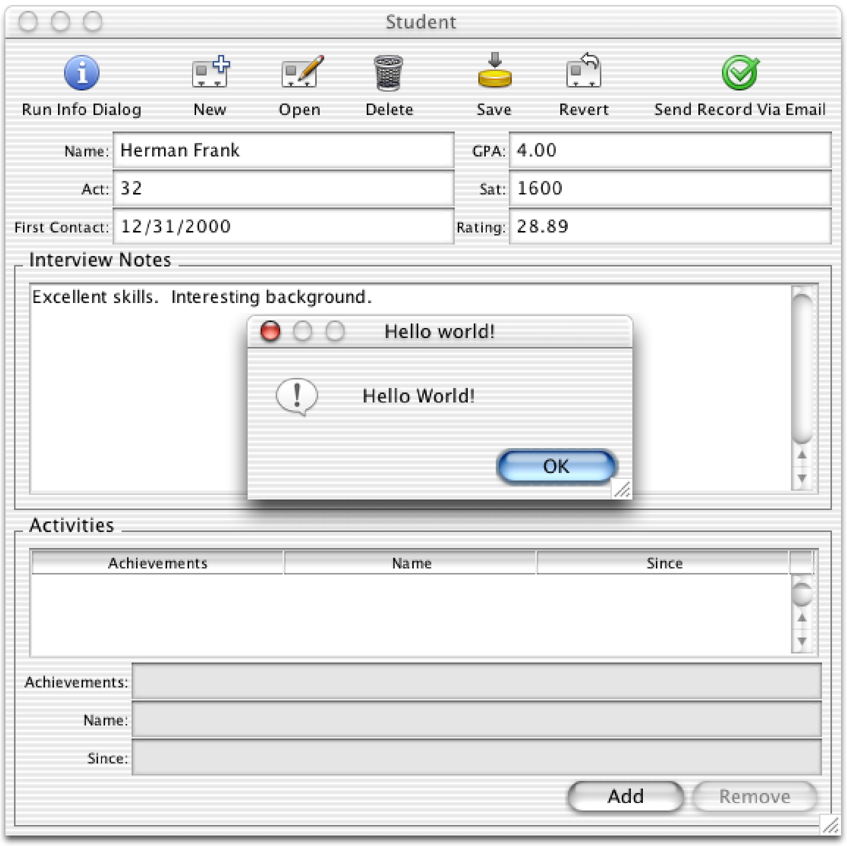
After writing the custom controller class, you must include it in a frozen XML component:

```
<CONTROLLER className="businesslogic.client.NewController">
```

Make sure to place it in the XML hierarchy within an `ACTIONBUTTONSCONTROLLER` block.

The implementation of the action method in this example simply puts up a dialog as shown in Figure 16-7 (page 208).

Figure 16-7 Action in custom controller class



Name: Herman Frank
GPA: 4.00
Act: 32
Sat: 1600
First Contact: 12/31/2000
Rating: 28.89

Interview Notes

Excellent skills. Interesting background.

Hello world!

Hello World!

Mixing Static and Dynamic User Interfaces

In this chapter, you'll learn how to integrate static interfaces made in Interface Builder with dynamically generated user interfaces.

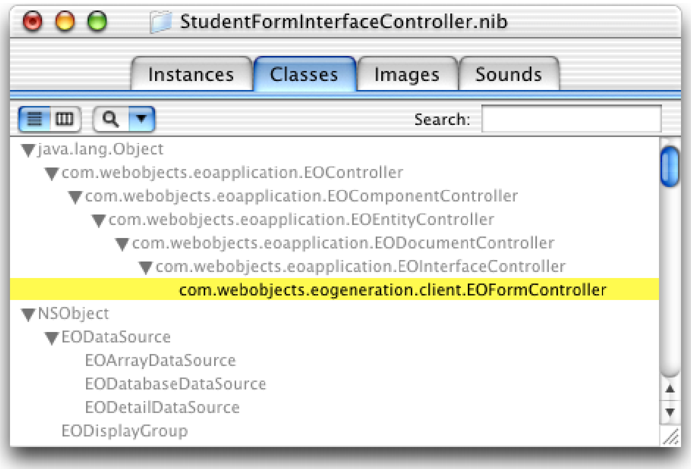
For this chapter you need a nib file. Any one will do, so you can use one from one of the WebObjects example projects or one you've created. It's also sufficient to just add a Java Client Interface file to your project and add some widgets to it. Whatever nib file you use, make sure to first add it to your project and associate it with the Web Server target.

Preparing the Nib for Freezing

You must do a few things to make interface files created with the nondirect Java Client technique work within dynamically generated user interfaces.

Open the nib file from within Project Builder and click the Classes tab of the nib file window. View the classes in inheritance mode (the vertical list), and click the disclosure triangle next to `java.lang.Object` to reveal the Java Client classes. Continue clicking disclosure triangles up through `com.webobjects.eoapplication.EOInterfaceController` as shown in Figure 17-1.

Figure 17-1 Classes pane in the nib file window



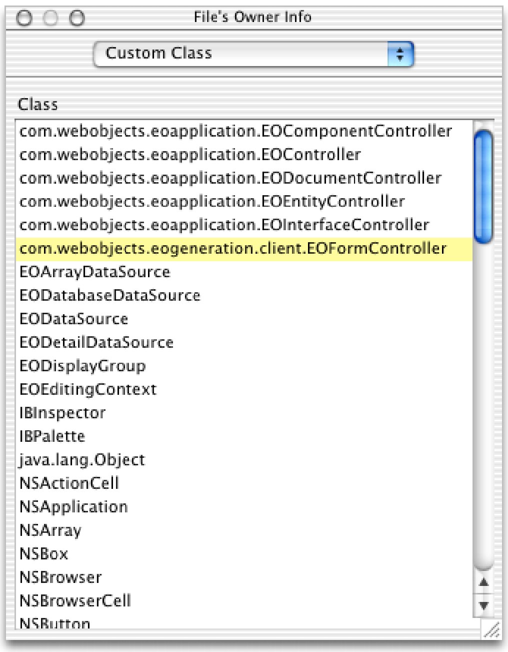
```
▼java.lang.Object
  ▼com.webobjects.eoapplication.EOController
    ▼com.webobjects.eoapplication.EOComponentController
      ▼com.webobjects.eoapplication.EOEntityController
        ▼com.webobjects.eoapplication.EODocumentController
          ▼com.webobjects.eoapplication.EOInterfaceController
              com.webobjects.eogeneration.client.EOFormController
▼NSObject
  ▼EODataSource
      EOArrayDataSource
      EODatabaseDataSource
      EODetailDataSource
    EODisplayGroup
```

To use a nib file in a Direct to Java Client application, the class must be the exact class you use to load the interface. So, if you want to use the nib file in a form controller, you'll use EOFormController. To use it in a query controller, use EOQueryController. These classes are not automatically defined in the EnterpriseObjects palette in Interface Builder, so you need to add them.

Select `com.webobjects.eoapplication.EOInterfaceController` in the classes list and press Return. This subclasses EOInterfaceController and thus the new class inherits the targets and outlets you need for it. Name the new subclass `com.webobjects.eogeneration.EOFormController` as shown in Figure 17-1 (page 209).

Now that you've created a new class, you must associate the nib file with it. To do this, go back to the Instances pane of the nib file window and click File's Owner. Choose Show Info from the Tools menu and choose Custom Class from the pop-up menu. In the list of classes, select `com.webobjects.eogeneration.EOFormController` as shown in Figure 17-2.

Figure 17-2 Assign the custom subclass to File's Owner



com.webobjects.eoapplication.EOComponentController
com.webobjects.eoapplication.EOController
com.webobjects.eoapplication.EODocumentController
com.webobjects.eoapplication.EOEntityController
com.webobjects.eoapplication.EOInterfaceController
com.webobjects.eogeneration.client.EOFormController
EOArrayDataSource
EODatabaseDataSource
EODataSource
EODetailDataSource
EODisplayGroup
EOEditingContext
IBInspector
IBPalette
java.lang.Object
NSActionCell
NSApplication
NSArray
NSBox
NSBrowser
NSBrowserCell
NSButton

Save the nib file.

Finally, associate the nib file's controller class (its associated `.java` class) with the same package with which other client-side classes in your application are associated:

```
package edu.admissions.client;
```

Integrating the Nib File

The nib file is now ready to be integrated into a dynamically generated Java Client user interface. To load it in an application, you need to write a rule.

Open the `d2w.d2wmodel` file from within the your project. The rule shown here assumes that you want to use the nib file in a form controller for an entity named "Student," that the nib file is named `StudentFormInterfaceController`, and that it's in the package `edu.admissions.client`.

Left-Hand Side:	`((task = 'form') and (entity.name = 'Student') and (controllerType = 'entityController'))`
Key:	`archive`
Value:	`"edu.admissions.client.StudentFormInterfaceController"`
Priority:	`50`

This rule says that for the form task for the Student entity, use an archive (a nib file) with the name `StudentFormInterfaceController`. So, when you make a new Student record, the nib file is loaded.

However, at this point the XML-based interface generated by the rule system is loaded. Just because you're loading a nib file does not suppress the mechanism for generating the interface's subcontrollers. But, it's easy to write a rule to fix this:

Left-Hand Side:	`((task = 'form') and (entity.name = 'Student') and (controllerType = 'entityController'))`
Key:	`generateSubcontrollers`
Value:	`"false"`
Priority:	`50`

Now when you open a form window for the Student entity, the custom interface is loaded and the XML generation is suppressed in certain parts of the window.

Using Nib Actions When Mixing

When you use nib files within dynamically generated user interfaces as described in "Mixing Static and Dynamic User Interfaces" (page 209), the nib file's controller class changes. This means that the class in which you declare actions that are the target of widgets in the nib file changes.

When you use nib files in nondirect Java Client applications, the controller class of each nib is an EOInterfaceController subclass that has the same name as the nib. This means that the File's Owner object in the nib file is associated with this controller class. So, if a widget such as a button has a target that is a method called `search`, that method must be declared in the EOInterfaceController subclass that is the object with which File's Owner is associated.

As described in "Integrating the Nib File" (page 210), however, to use a nib file within a dynamically generated user interface, you need to change the nib file's controller class to be the class that is the root controller class of the dynamically generated user interface.

So, for example, if you are integrating a nib file into a form window, as described in "Integrating the Nib File" (page 210), that window's root class is EOFormController. So the target methods of actions in that window, whether the actions originate from widgets in the nib file or widgets in the dynamically generated user interface, must be in the EOFormController subclass.

In summary, when you use a nib file within a dynamically generated user interface, the nib file's original controller class (a subclass of EOInterfaceController) is no longer used so you must move any action methods out of that class and into the controller class of the user interface the nib file is used within (such as EOFormController and EOQueryController).

Using Custom Views in Nib Files

The Java Client interfaces you can build in Interface Builder support only a subset of all the standard Swing components. However, by using custom views in interface files, you can use any Swing component or custom components you write. This chapter describes how to use custom views in interface files and then provides some examples of custom view components.

You can use the nib file you built in "Nondirect Java Client Development" (page 137) for the exercises in this chapter.

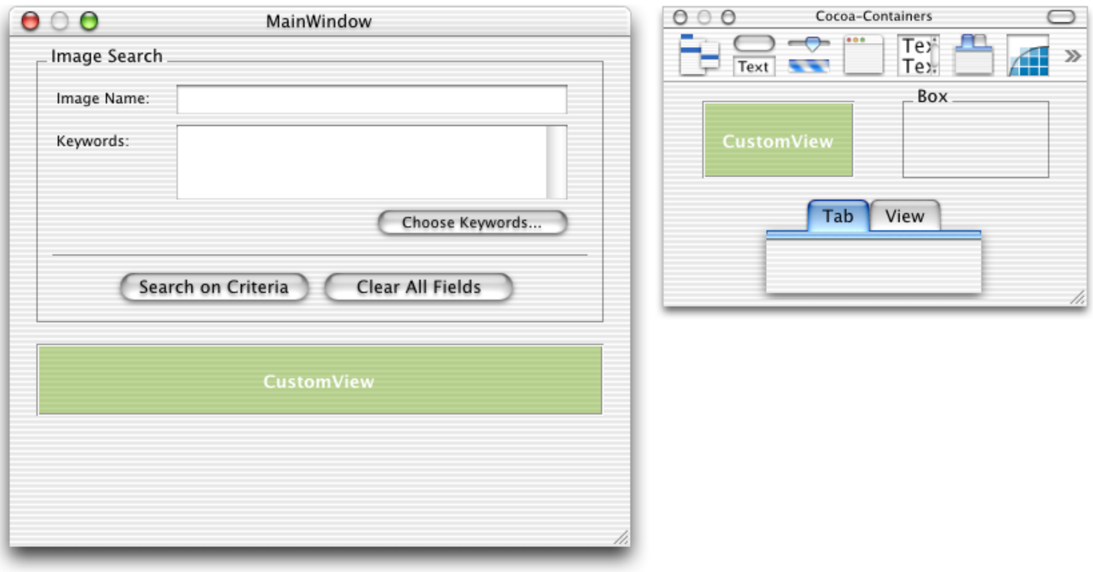
Custom Views

Problem: You want to add an unsupported view in an interface file such as `javax.swing.JProgressBar`.

Solution: Place a custom view object and connect it to an outlet in File's Owner.

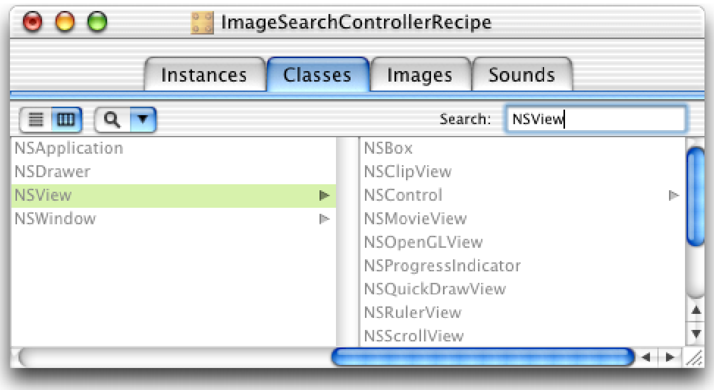
In an Interface Builder file, place a custom view object in the main window. You can find this object in the Containers palette. Figure 18-1 shows this palette and a custom view placed in the main window.

Figure 18-1 Custom view object in window



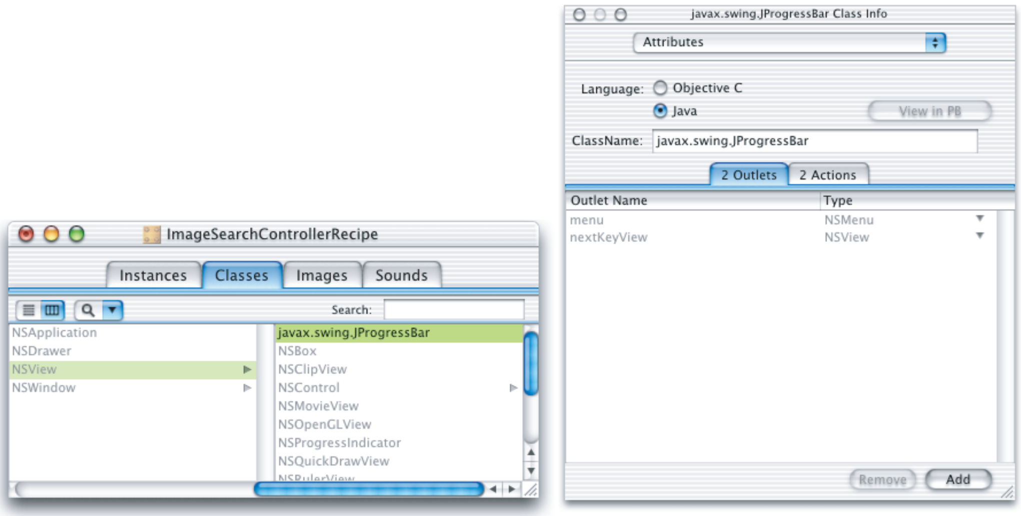
Next, you need to assign the custom view to an NSView subclass. Before you can do this, you need to create an NSView subclass. Switch to the Classes pane in the nib file window and enter `NSView` in the Search field as shown in Figure 18-2.

Figure 18-2 Find NSView in class hierarchy



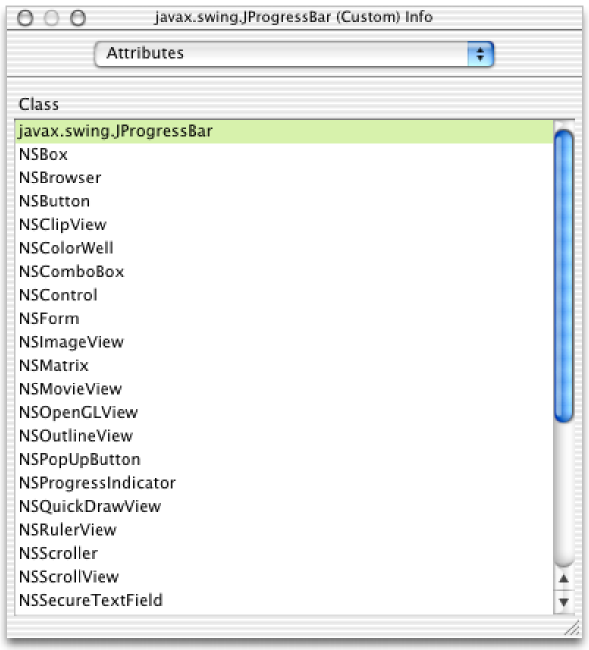
Select NSView if it is not already selected and press Return to subclass it. In the Info window, select Java as the language for the subclass. Then, provide a fully qualified name for the subclass. If the view represents a Swing class such as JProgressBar, use `javax.swing.JProgressBar` as shown in Figure 18-3. If the view represents a custom Swing subclass, specify the fully qualified name of that subclass.

Figure 18-3 Name the custom view class



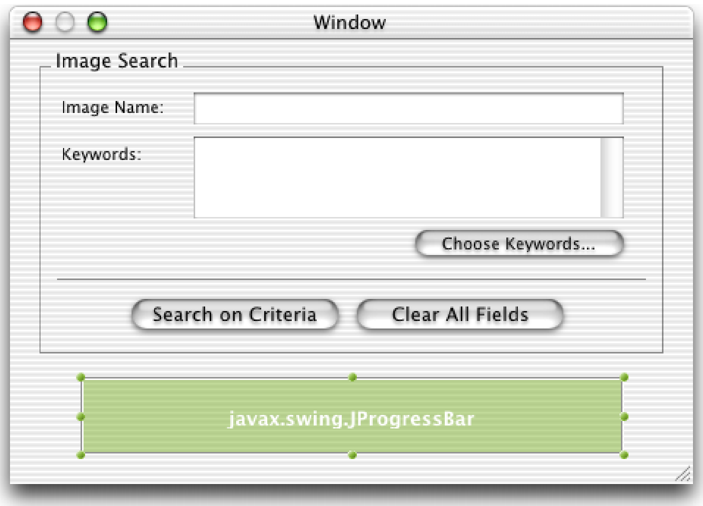
Next, you need to associate the custom view you placed in the window with the new NSView subclass. Select the custom view widget in the main window and bring up the Attributes pane of the Info window. Select `javax.swing.JProgressBar`, as shown in Figure 18-4.

Figure 18-4 Associate custom view with NSView subclass



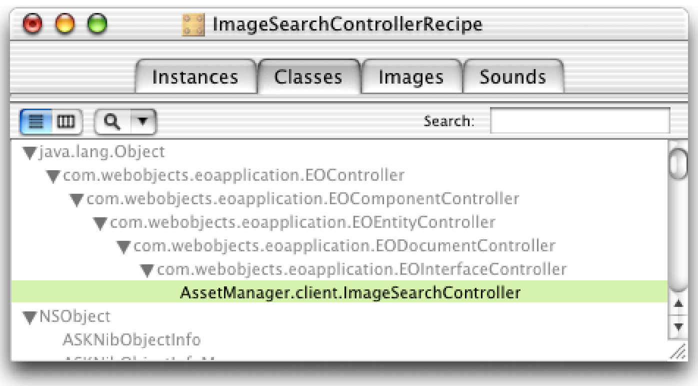
The name in the custom view should then change to the name of the new class, as shown in Figure 18-5.

Figure 18-5 Custom view as NSView subclass



Now you need to add an outlet to the interface file's File's Owner object for the custom view. This gives you programmatic access to the widget in the nib file's controller class, which allows you to query and change the widget's attributes. In the Classes pane of the nib file window, view the class hierarchy vertically and disclose the list starting with `java.lang.Object` as far as you can, as shown in Figure 18-6.

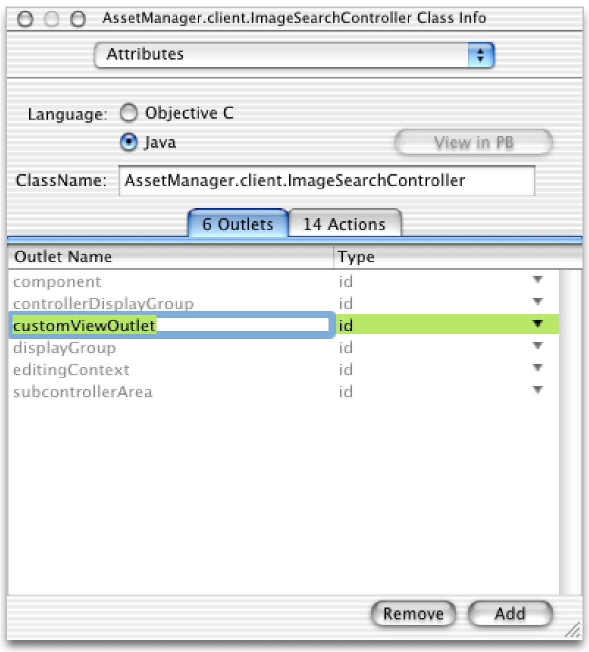
Figure 18-6 File's Owner class



```
▼java.lang.Object
  ▼com.webobjects.eoapplication.EOController
    ▼com.webobjects.eoapplication.EOComponentController
      ▼com.webobjects.eoapplication.EOEntityController
        ▼com.webobjects.eoapplication.EODocumentController
          ▼com.webobjects.eoapplication.EOInterfaceController
              AssetManager.client.ImageSearchController
▼NSObject
    ASKNibObjectInfo
```

Select the last class in the hierarchy and bring up the Info window. Add an outlet to the class called `customViewOutlet`, as shown in Figure 18-7.

Figure 18-7 Add outlet to interface file



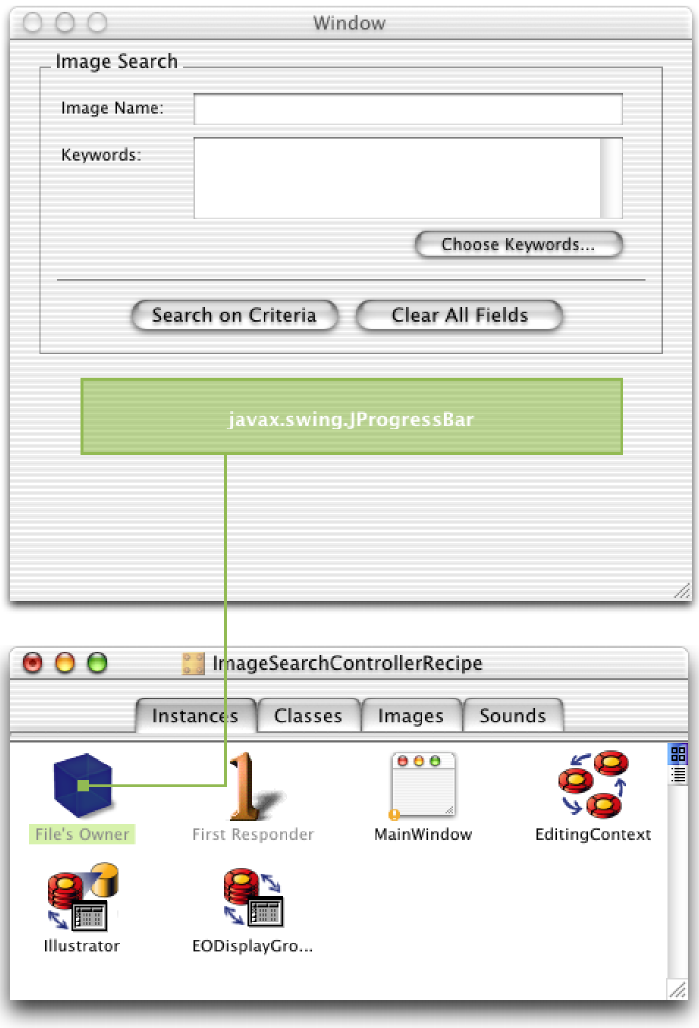
Language: ○ Objective C [✓ Java]

ClassName: AssetManager.client.ImageSearchController

Outlet Name	Type
component	id
controllerDisplayGroup	id
customViewOutlet	id
displayGroup	id
editingContext	id
subcontrollerArea	id

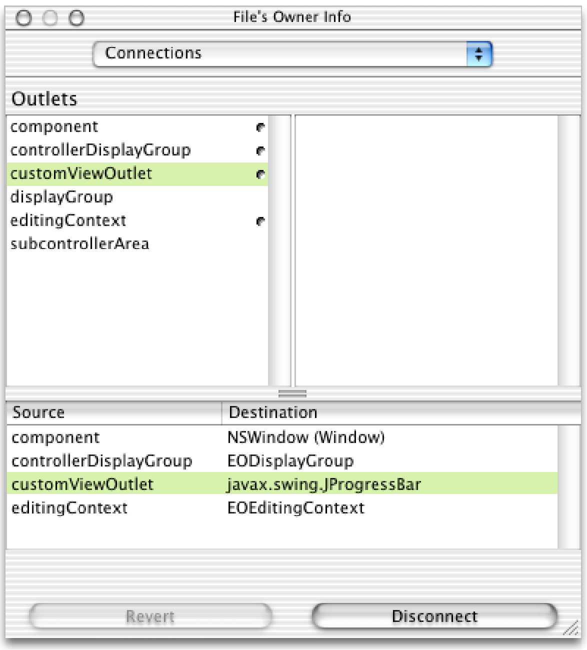
Next, you need to connect the custom view to the outlet you just created. Switch to the Instances pane of the nib file window and Control-drag from File's Owner to the custom view in the main window as shown in Figure 18-8.

Figure 18-8 Connect new outlet to custom view



Then in the Connections pane of the Info window, select `customViewOutlet` and click Connect. The Connections pane of the Info window for File's Owner should now appear as shown in Figure 18-9.

Figure 18-9 File's Owner attributes



component
controllerDisplayGroup
customViewOutlet
displayGroup
editingContext
subcontrollerArea

Source	Destination
component	NSWindow (Window)
controllerDisplayGroup	EODisplayGroup
customViewOutlet	javax.swing.JProgressBar
editingContext	EOEditingContext

Save the interface file and open its controller class (`.java` file) in Project Builder. Add an instance variable for the outlet you added:

```
public JProgressBar customViewOutlet;
```

You now have a JProgressBar widget in your interface file. You can set its value by invoking `customViewOutlet.setValue(int value)` in the controller class. However, don't attempt to invoke methods on the widget in the interface controller's constructors as it may not be initialized at that point. Rather, override `controllerDidLoadArchive` as described in "Loading the Image" (page 224) or check to see if the component is initialized by invoking `isComponentPrepared`.

Make EOImageView Accept Clicks

You can apply what you learned in the last section to extend the power of the user interface components supplied by the `com.webobjects.eointerface.swing` package. This section describes how to extend the EOImageView class to support mouse clicks.

Problem: The class `com.webobjects.eointerface.swing.EOImageView` does not support mouse clicks.

Solution: Subclass EOImageView and provide custom view outlets in an Interface Builder nib file or write a rule to use the subclass in certain controllers.

To make an EOImageView object respond to mouse clicks, you need to subclass MouseAdaptor within an EOImageView subclass. Add a file to your project named `CustomImageViewController`. Paste this code into it:

```
package com.mycompany.myapp;

import java.awt.*;
import javax.swing.event.*;
import com.webobjects.foundation.*;
import com.webobjects.eointerface.swing.*;
import com.webobjects.eogeneration.*;

public class CustomImageViewController extends EOImageView {

    public CustomImageViewController() {
        super();
        this.addMouseListener(new OpenRecord());
    }

    class OpenRecord extends MouseInputAdapter {

        public void mouseClicked(MouseEvent e) {
            NSLog.out.appendln("image clicked");
        }

    }

}
```

To use this custom class in an interface file, you subclass NSView as described in "Custom Views" (page 213) and name the subclass `com.mycompany.myapp.CustomImageViewController`.

Using and Extending Image Views in Nib Files

It's common to want to display images in the user interfaces of Java Client applications. Although you can use Interface Builder to place the view area for an image, you must retrieve and load the image programmatically. This task describes the steps necessary to use an image view in an Interface Builder file.

Problem: You want to display an image in an Interface Builder file.

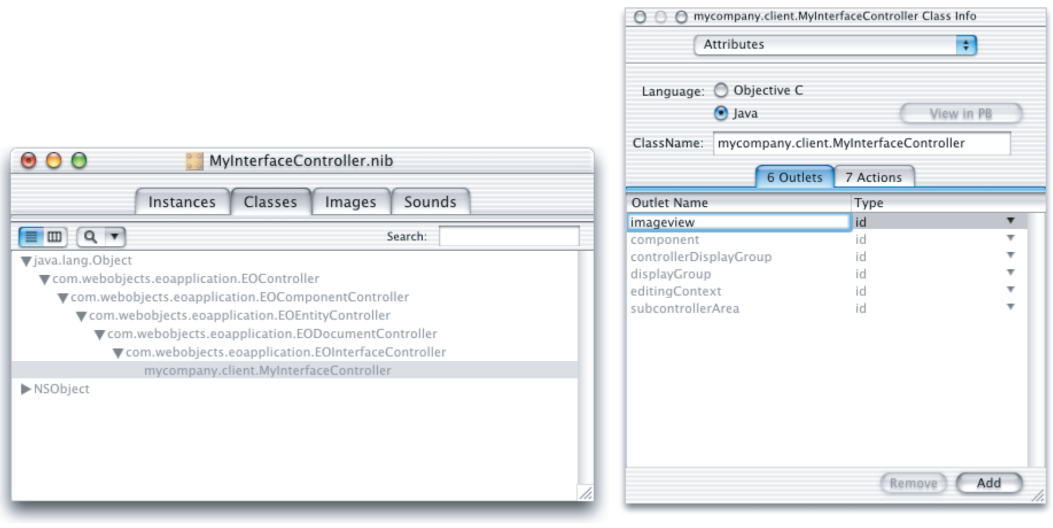
Solution: Place an image view in a nib file, add an outlet, and load the image programmatically.

Adding Outlets

To set the image in an image view, you need access to the widget in the controller class. This is described in "Programmatic Access to Interface Components" (page 151). To review, to get programmatic access to user interface elements, you need to add outlets to File's Owner, associate user interface widgets with those outlets, and add instance variables for the outlets to which you want programmatic access.

To add an outlet, switch to the Classes pane in the nib file window and select the class with which File's Owner is associated. Bring up the Info window and choose Attributes from its pop-up list. Switch to the Outlets pane and click Add to add a new outlet. Name the new outlet `imageview`. Refer to Figure 19-1.

Figure 19-1 Add a new outlet



```
▼java.lang.Object
  ▼com.webobjects.eoapplication.EOController
    ▼com.webobjects.eoapplication.EOComponentController
      ▼com.webobjects.eoapplication.EOEntityController
        ▼com.webobjects.eoapplication.EODocumentController
          ▼com.webobjects.eoapplication.EOInterfaceController
              mycompany.client.MyInterfaceController
▶NSObject
```

Language: Objective C [✓ Java]

ClassName: mycompany.client.MyInterfaceController

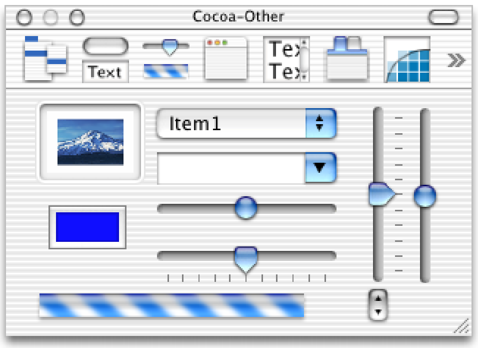
Outlet Name	Type
imageview	id
component	id
controllerDisplayGroup	id
displayGroup	id
editingContext	id
subcontrollerArea	id

Now you're ready to add the image view widget to the interface.

Adding the Widget

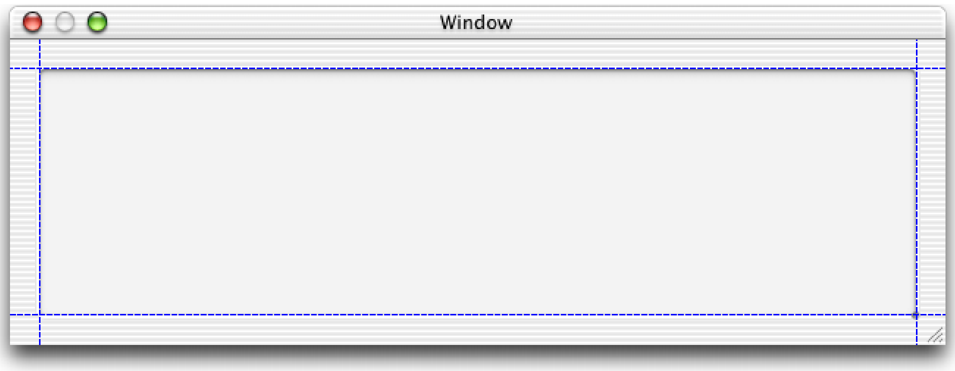
You now need to add a widget to represent the image you want to display. The Other palette includes an image view widget that works for these purposes. If the palettes window isn't visible, choose Tools >Palettes > Show Palettes. Click the Other palette button in the toolbar (the one with the slider and the progress bar). The Other palette is shown in Figure 19-2.

Figure 19-2 Other palette



Drag the image view widget (the one in the upper-left corner of the Cocoa palette with the picture of a mountain in it) onto the main window. If the main window isn't visible, switch to the Instances pane of the nib file window and double-click the MainWindow object. Place the widget in the upper-left corner of the window and use the guides Interface Builder provides to size and place it, as shown in Figure 19-3.

Figure 19-3 Place widget with guides

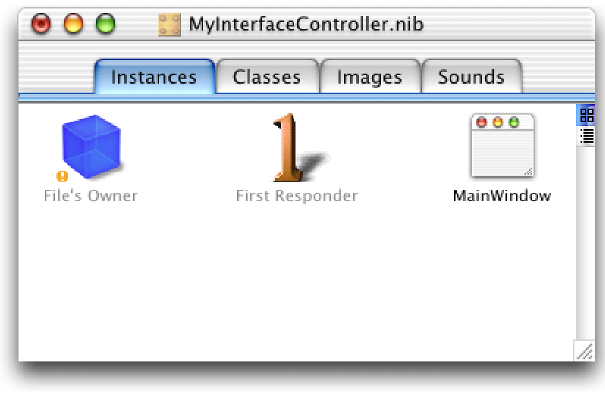


Now you need to connect the widget to the outlet you added to File's Owner.

Connecting the Outlet

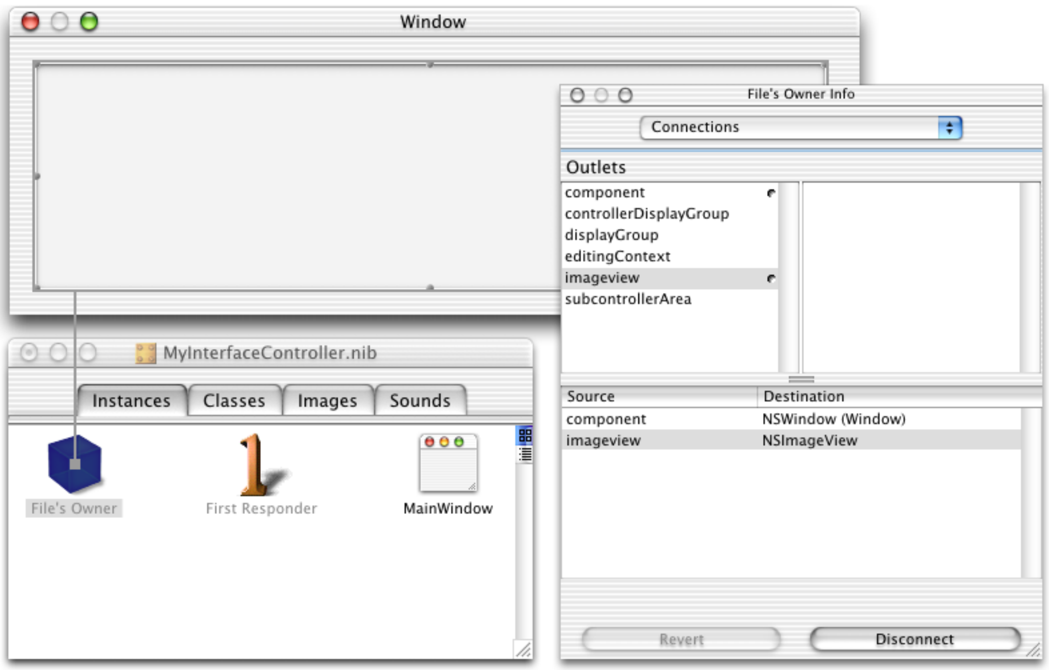
Interface Builder is the best tool for building Java Client user interfaces as it allows you to visually associate user interface widgets with outlets and actions in the class file. When you associated File's Owner with the custom subclass of EOInterfaceController, the icon for File's Owner changed to include an exclamation point, as shown in Figure 19-4.

Figure 19-4 File's Owner icon with exclamation point



The exclamation point icon indicates that File's Owner's connections are broken or incomplete. In this case, the `imageview` outlet you added is not connected (it is not associated with anything). To make the connection, Control-drag from File's Owner to the image view widget you placed in the main window as shown in Figure 19-5.

Figure 19-5 Connect outlet to widget



component
controllerDisplayGroup
displayGroup
editingContext
imageview
subcontrollerArea

Source	Destination
component	NSWindow (Window)
imageview	NSImageView

In the File's Owner Info window, select the `imageview` outlet and click Connect. The File's Owner Info window should then appear as in Figure 19-5.

Save the interface file. It is now prepared to display an image in the client application.

Loading the Image

The image view widget you placed in the interface file did not specify a particular image. Rather, it specified an area in the window where an image can be displayed. You now need to add some code to retrieve the image and load it in the image view widget.

In Project Builder, open the Java class for the interface file you just edited. Before modifications, it should look something like this:

```
package mycompany.client;

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

public class MyInterfaceController extends EOInterfaceController {

    public MyInterfaceController() {
        super();
    }
    public MyInterfaceController(EOEditingContext substitutionEditingContext)
{
        super(substitutionEditingContext);
    }
}
```

To load and place the image, you use the EOInterface Swing package (`com.webobjects.eointerface.swing.*`). You also need to use the Java Swing and AWT packages, so add these import statements:

```
import com.webobjects.eointerface.swing.*;
import java.awt.*;
import javax.swing.*;
```

Next, you need to add an instance variable to get access to the outlet you defined in the interface file. The variable's type is EOImageView, defined in `com.webobjects.eointerface.swing.*`. You named the outlet "imageview" so add an instance variable of the same name:

```
public EOImageView imageview;
```

To load an image into the EOImageView object, you need to make sure that the interface controller has finished loading. The method `controllerDidLoadArchive` is invoked when the controller is finished loading. You can override it to perform certain initializations, such as loading an image into an image view. Add the method as shown in Listing 19-1.

Listing 19-1 Overriding `controllerDidLoadArchive`

```
    protected void controllerDidLoadArchive(NSDictionary namedObjects) {

        ImageIcon iIcon =                                                // 1
(ImageIcon)EOUserInterfaceParameters.localizedIcon("iMac");

        Image newImage = iIcon.getImage();                               // 2

        imageview.setImage(newImage);                                    // 3
    }
```

Code line 1 attempts to retrieve an image that is associated with the Web Server target. Specify the image name without including the suffix.

Note: As of WebObjects 5.2, `EOUserInterfaceParameters.localizedIcon` retrieves images with extensions `gif`, `jpeg`, `jpg`, and `png` only.

Code line 1 casts the retrieved object into an object of type ImageIcon. `localizedIcon` returns an object of type Icon, so casting the retrieved object into an ImageIcon allows you to retrieve the image data in the form of an Image object that the `setImage` method on EOImageView accepts. Code line 2 retrieves the image's data from the ImageIcon object and code line 3 sets the image in the image view object to the image retrieved in code line 1.

If successful, your interface file should load and display the specified image as shown in Figure 19-6.

Figure 19-6 Image in image view

Using Pop-up Menus in Nib Files

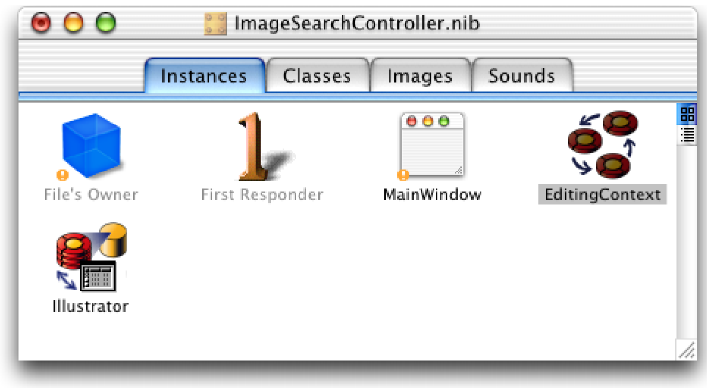
It's common to want to display pop-up menus in interface files that display a short list of enumeration values. This chapter describes how to connect a pop-up menu widget (`javax.swing.JComboBox`) to a display group and how to get the value of the selected object in the interface file's controller class.

Problem: You want to display a pop-up menu (JComboBox) and extract the selected value.

Solution: Place a pop-up menu widget in an interface file and use a controller display group to extract the value.

In a nib file, add the entity that contains the enumeration values to the nib file by dragging the entity from EOModeler into the nib file window. Figure 20-1 shows an entity called "Illustrator" as a display group in a nib file.

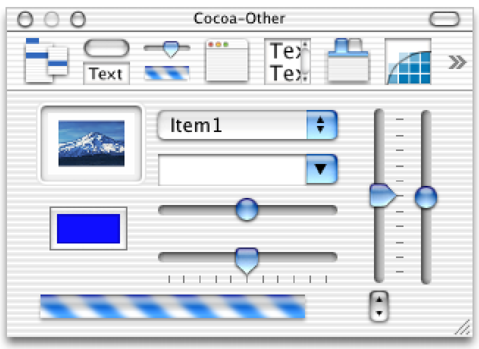
Figure 20-1 Illustrator entity in nib file



When you drag an entity from EOModeler into a nib file, an EOEditingContext object is also added if one is not already in the nib file.

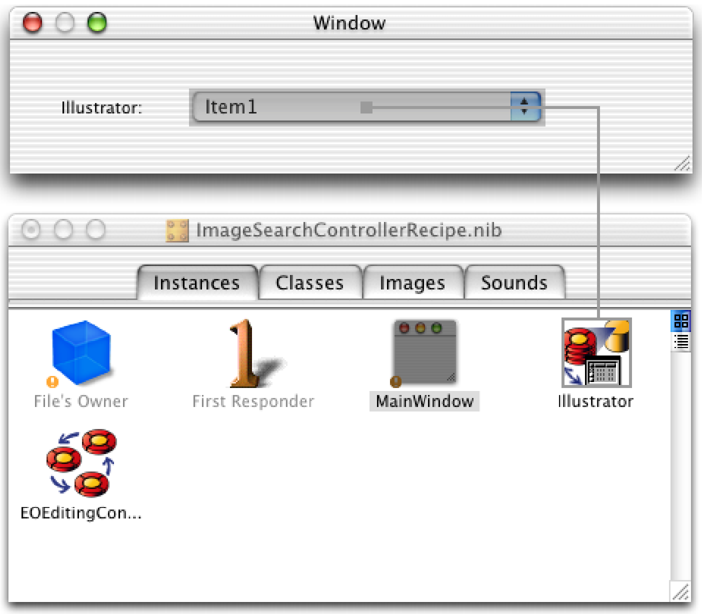
Now add a widget for the pop-up menu. You can find it in the Other palette, as shown in Figure 20-2. It's the widget that includes the text "Item1".

Figure 20-2 Other palette



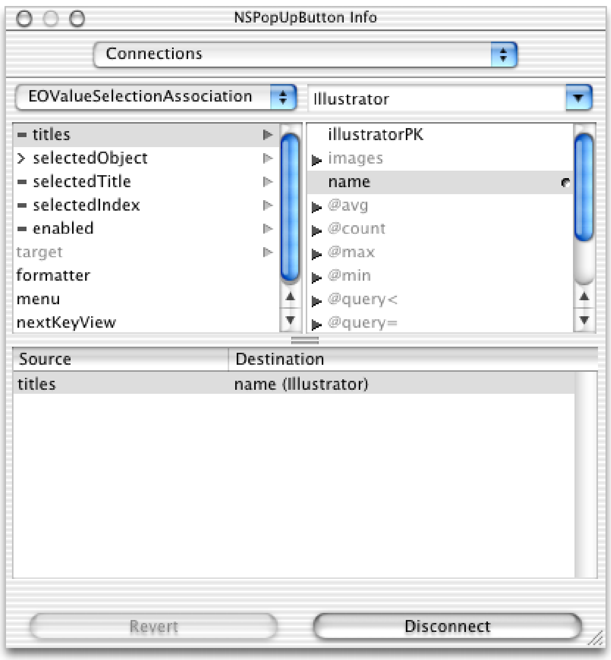
Then, Control-drag from the widget to the display group for the entity containing the enumeration values, as shown in Figure 20-3.

Figure 20-3 Connect widget to display group



This action displays the Info window so you can set the binding for the `titles` aspect of the EOValueSelectionAssociation. As shown in Figure 20-4, bind the `titles` aspect to the attribute of the entity that represents the enumeration value, `name` in the example shown here.

Figure 20-4 Bind the title aspect to the appropriate attribute

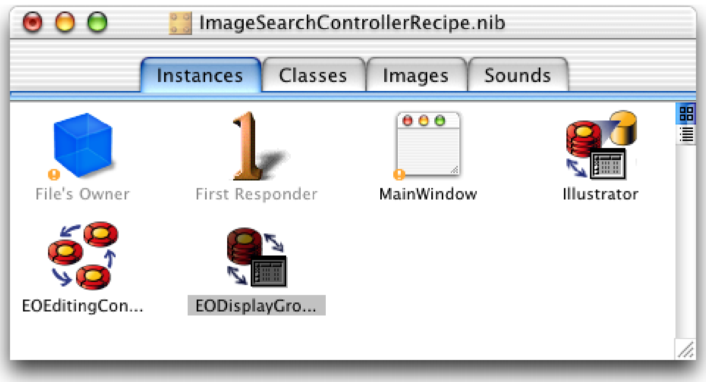


Source	Destination
titles	name (Illustrator)

Save the nib file and choose Test Interface from the File menu. You should see the values of the attribute bound to the `titles` aspect of the pop-up menu as items in that menu.

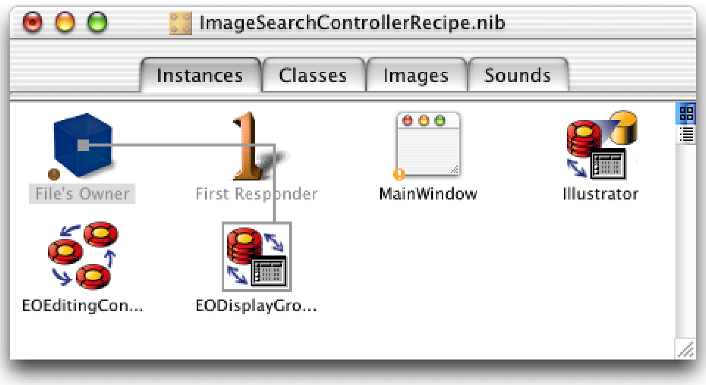
To get the value of the selected object in the controller class for the interface file, there is more work to do. Add a new EODisplayGroup object to the interface file by dragging one out from the EnterpriseObjects palette into the nib file window. The nib file window should then appear as shown in Figure 20-5.

Figure 20-5 EODisplayGroup object in nib file



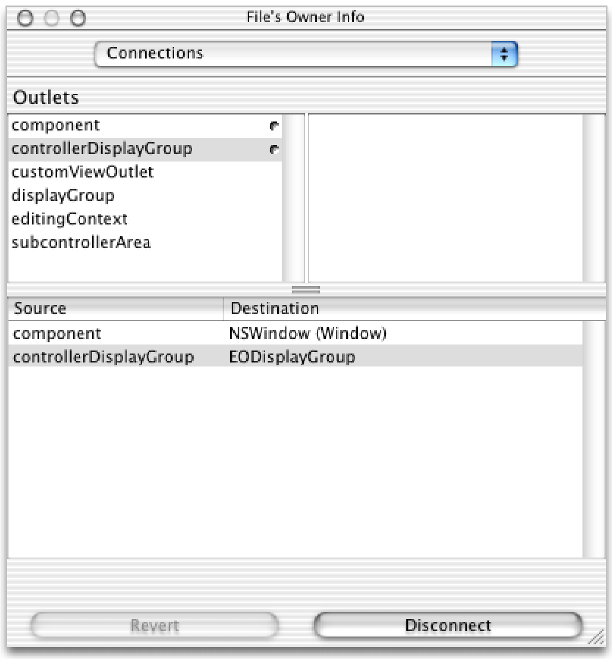
Then, bind the new EODisplayGroup object to the `controllerDisplayGroup` outlet of File's Owner. Do this by Control-dragging from File's Owner to the new display group as shown in Figure 20-6.

Figure 20-6 Bind File's Owner `controllerDisplayGroup` outlet



Then, in the Info window, select `controllerDisplayGroup` and click Connect, as shown in Figure 20-7.

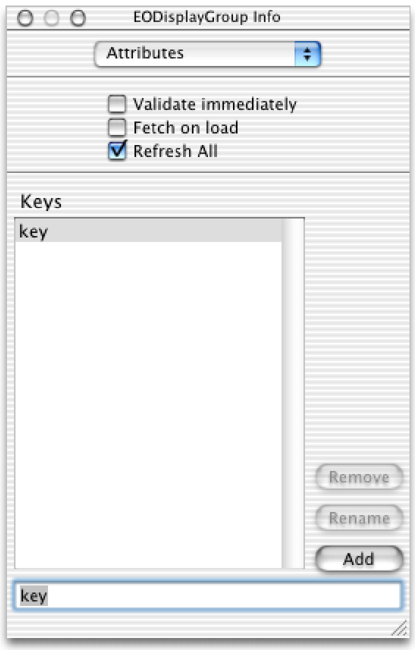
Figure 20-7 Bind the outlet



Source	Destination
component	NSWindow (Window)
controllerDisplayGroup	EODisplayGroup

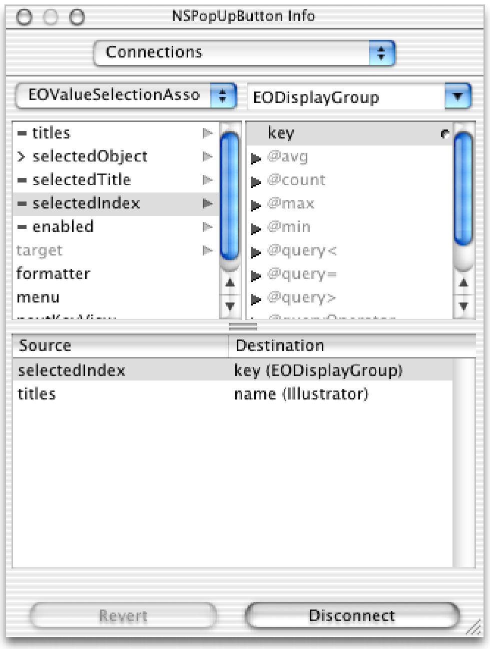
Now, add a key to the controller display group object called `key`. This represents the name of the action method that is invoked in the nib file's controller class when a user chooses an object in the pop-up menu. To add a key, select the display group object in the nib file window and choose Show Info from the Tools menu. In the Attributes pane, add the key named `key` as shown in Figure 20-8.

Figure 20-8 Add a key to display group



Then, bind the `selectedIndex` attribute of the EOValueSelectionAssociation to the key named `key` in the controller display group. Control-drag from the pop-up menu to the display group bound to the `controllerDisplayGroup` outlet of File's Owner and in the Info window, connect the binding as shown in Figure 20-9.

Figure 20-9 Bind `selectedIndex` attribute of association to display group key



Source	Destination
selectedIndex	key (EODisplayGroup)
titles	name (Illustrator)

Save the nib file.

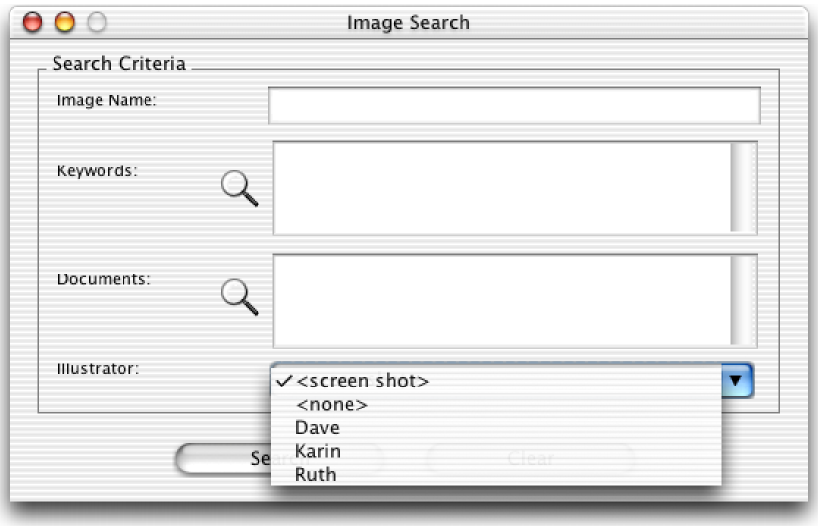
In the nib file's controller class, add a method called `setKey`. This is invoked when an object in the pop-up menu is selected.

```
public void setKey(int illustrator) {
     _illustrator =
(String)controllerDisplayGroup().valueForObjectAtIndex(illustrator,
"name");
}
```

It sets an instance variable in the class (`_illustrator`) to the String value of the object selected in the menu.

Figure 20-10 shows a pop-up menu in action.

Figure 20-10 A pop-up menu in action



Localizing Dynamic Components

Localization can be a tedious and time-consuming part of application development. However, using the rule system in Java Client applications, localization is quite simple. You supply a Java class containing the localized strings and you write a rule to use the class for labels in dynamically generated user interfaces.

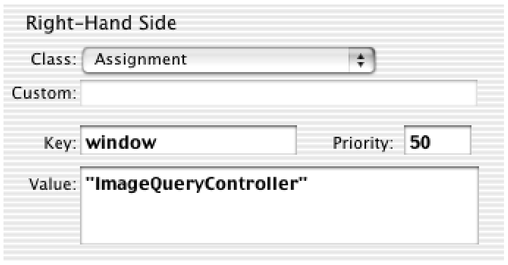
Localizing Property Labels

Problem: You want to localize the labels of properties in your application.

Solution: Write a Java class to perform the localized string lookup, get the user's preferred languages, and write a rule to get the localized strings.

Most of the rules you write and use in the rule system have a right-hand side class of type Assignment as shown in Figure 21-1.

Figure 21-1 Right-hand side class of type Assignment


Right-Hand Side

Class: Assignment

Custom:

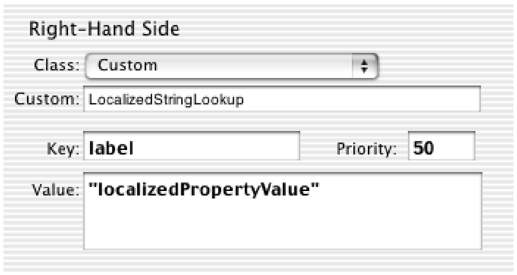
Key: window

Priority: 50

Value: "ImageQueryController"

The rule you'll write to localize dynamic components uses the type Custom. By specifying a class name in the Custom field and a method name in the Value field, the key specified in the Key field is assigned to the return value of the specified method in the specified class. In Figure 21-2, the key `label` is resolved to the result of the method named `localizedPropertyValue` in the class LocalizedStringLookup.

Figure 21-2 Right-hand class of type Custom


Right-Hand Side

Class: Custom

Custom: LocalizedStringLookup

Key: label

Priority: 50

Value: "localizedPropertyValue"

Before writing the rule, however, write the class that does the localized string lookup.

Add a class to your project called "LocalizedStringLookup." Add it to the Application Server target. Copy and paste the code in Listing 21-1.

Listing 21-1 LocalizedStringLookup class

```
import com.webobjects.foundation.*;
import com.webobjects.appserver.*;
import com.webobjects.eocontrol.*;
import com.webobjects.directtoweb.*;

public class LocalizedStringLookup extends DefaultAssignment {

    /**
     *Used in this class to refer to the current D2WContext object.
     */
    D2WContext d2wcontext;

    public LocalizedStringLookup(EOKeyValueUnarchiver unarchiver) {
        super(unarchiver);
    }
    public LocalizedStringLookup(String key, String value) { super(key,value); }

    public static Object decodeWithKeyValueUnarchiver(EOKeyValueUnarchiver
   eokeyvalueunarchiver)  {
        return new LocalizedStringLookup(eokeyvalueunarchiver);
    }

    /**
     *Since there is no public API to get the current D2WContext,override this method
       to get a reference to the current D2Wcontext.
     */
    public synchronized Object fire(D2WContext context) {
        d2wcontext = context;
        Object result = KeyValuePath.valueForKeyOnObject((String) value(), this);
        return result;
    }

    public String localizedPropertyValue() {
        String displayName = (String)d2wcontext.valueForKey(D2WModel.PropertyKeyKey);  // 1

        NSArray languages = (NSArray)d2wcontext.valueForKey("locales");

        String returnstr =                                                               // 2
WOApplication.application().resourceManager().stringForKey(displayName,
"Localizable", displayName, null, languages);

        return returnstr;

    }
}
```

Remember to change the package statement to the package your server-side (Application Server target) classes are in.

The most interesting part of the class is the `localizedPropertyValue` method. The rule you'll write invokes this method to get the localized string for a particular property. First, the method gets the display name for the receiver's property (code line 1). That is, if the property name is `date` (which corresponds to an attribute named `date` in an entity in one of the application's EOModels) the display name is the label that appears next to the widget representing the `date` property in the application.

Code line 2 is the most important part of the method. It looks for a localized string in a string table called `Localizable` for the display name specified by *`displayName`*. Since a localized application usually contains `Localizable.strings` files for multiple languages, the `stringForKey` method looks first for a `Localizable.strings` file for the user's first preferred language. If it finds a `Localizable.strings` file for that language, it returns the localized strings. If it does not, however, it continues through the user's preferred languages (returned by `d2wcontext.valueForKey("locales")`), defaulting to nonlocalized strings if it can't find a `Localizable.strings` file matching one of the user's preferred languages.

Now that you have the method to look up localized strings, you need to add localized string tables to your project.

First, add a new file to the Resources group of your project called `Localizable.strings`. Add it to the Application Server target. The syntax of a `Localizable.strings` file is rather simple:

```
{
    "propertyName" = "localizedString";
}
```

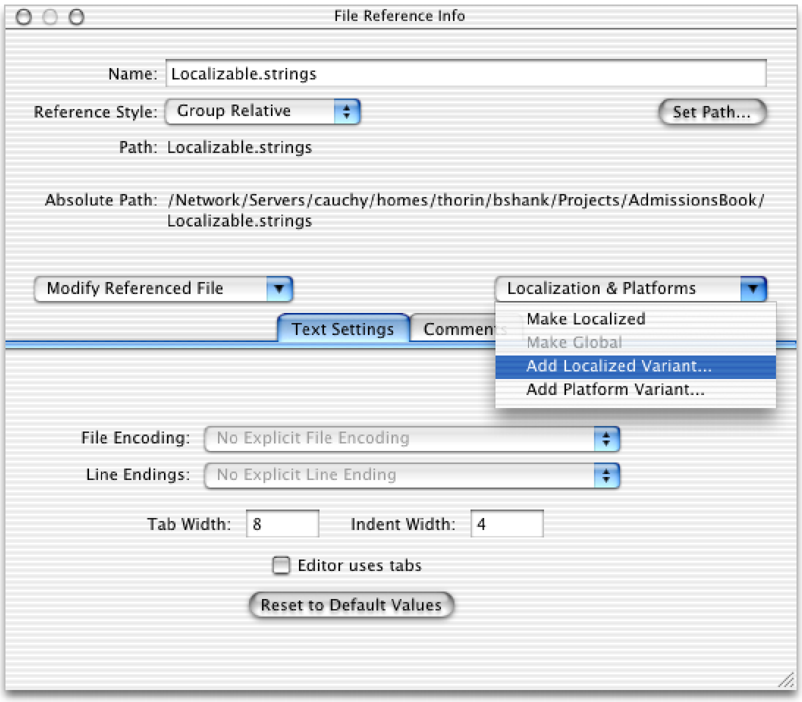
A `Localizable.strings` table for the property name "date" for Spanish would be

```
{
    "date" = "Fecha";
}
```

In the `Localizable.strings` table you just added to the project, add string pairs for the property keys in your application in English. You can find the names of the property keys in a few ways: in the Direct to Java Client Assistant's Properties pane; the output of the LocalizedStringLookup (which contains the log statement `NSLog.out.appendln("displayName: " + displayName);`); or by invoking `attributeKeys` on an enterprise object's class description and printing the result.

When you're done adding English-localized strings, you can add localized variants of the file to your project. Select the `Localizable.strings` file and choose Show Info from Project Builder's Project menu. From the Localization and Platforms pop-up menu, choose Add Localized Variant as shown in Figure 21-3.

Figure 21-3 Add localized variant of `Localizable.strings` file



Name: Localizable.strings

Reference Style: Group Relative

Path: Localizable.strings

Absolute Path: /Network/Servers/cauchy/homes/thorin/bshank/Projects/AdmissionsBook/Localizable.strings

File Encoding: No Explicit File Encoding

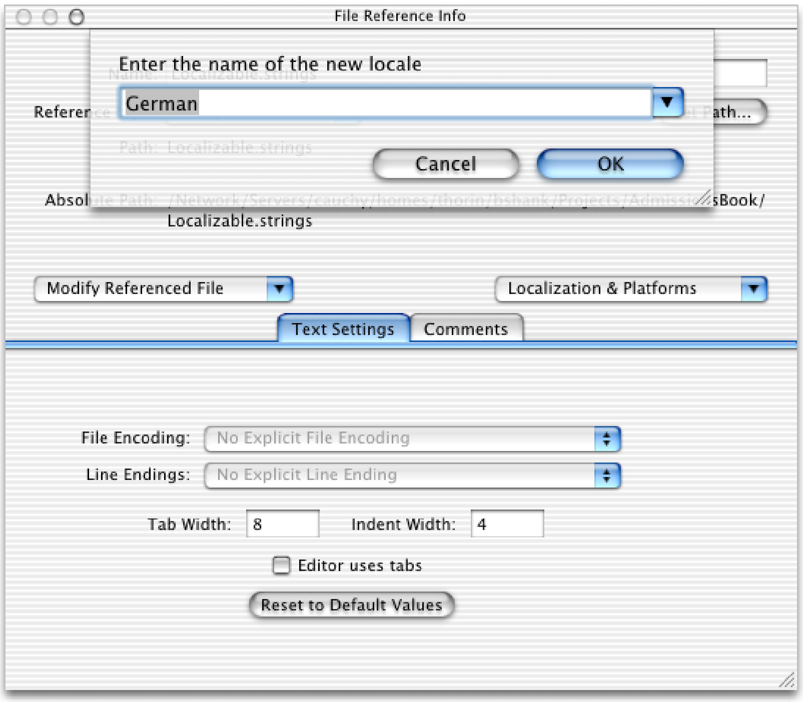
Line Endings: No Explicit Line Ending

Tab Width: 8 Indent Width: 4

Editor uses tabs

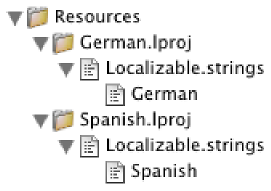
Add a localized variant for the language of your choice as shown in Figure 21-4. If the language is not listed, you can type it in the field underneath "Enter the name of the new locale."

Figure 21-4 Add localized variant for German



This action creates a directory called `German.lproj` (or whatever language you chose) in your project and puts a copy of the `Localizable.strings` file in it. Figure 21-5 shows German and Spanish localized variants in the Files list.

Figure 21-5 Localized resources in project



Now that you've created localized variants, you need to edit the variant to provide the language-specific strings for each property key. The German-localized variant might look like Listing 21-2 (page 239).

Listing 21-2 German-localized variants of strings file

```
{
    "modified" = "Geändert";
    "documents" = "Dokumente";
    "release" = "Freigeben";
    "keywords" = "Schlüsselwörter";
    "date" = "Datum";
    "notes" = "Anmerkungen";
    "illustrator" = "Illustrator";
}
```

Note: Make sure that the encoding for all `Localizable.strings` files in your project is Unicode 16 (full UTF-16, not UTF-8). You can change the encoding of a file by choosing an encoding from the File Encodings submenu of Project Builder's Format menu.

There is just one more thing you need to do to complete localization. Although the current process may seem tedious, think of the time it will save you: It saves you from needing to build localized variants of Interface Builder files by hand, or worse yet, from building localized versions of raw Swing components.

The final step is to write a rule to use everything you've just added to the application.

Left-Hand Side:	`*true*`
Key:	`label`
Class:	`Custom`
Custom:	`LocalizedStringLookup`
Value:	`"localizedPropertyValue"`
Priority:	`50`

The key `label` is assigned to the return value of the method `localizedPropertyValue` in the class LocalizedStringLookup. In Rule Editor, this rule appears as in Figure 21-2 (page 235).

Localizing the Standard Strings and Frozen XML Components

Problem: You want to localize all the standard strings in an application such as action button labels and standard error message strings. You also want to localize the property labels for frozen XML components.

Solution: Use similar localization techniques described in "Localizing Property Labels" (page 235), adding string pairs for each string you want localized.

First, associate the `Localizable.strings` files in your project with the Web Server target. (These files should now be associated with both the Web Server target and the Application Server target.) In "Localizing Property Labels" (page 235), you were instructed to associate these files with just the Application Server target. The localization technique described in that section happens in the server-side application. The technique described in this section, however, happens in the client-side application so the `.strings` files need to be made available to the client application.

By adding localized strings to the `Localizable.strings` files in each of your application's language `.lproj` directories, you can easily localize all the standard application strings. To find out what all these strings are, find the `Localizable.strings` file in `/System/Library/Frameworks/JavaEOApplication.framework/WebServerResources/English.lproj/`.

The string table begins with these string pairs:

```
"About Web Objects" = "About Web Objects";
"Actions" = "Actions";
"Activate Previous Window" = "Activate Previous Window";
"Add" = "Add";
"Add failed" = "Add failed";
"Append" = "Append";
"Append failed" = "Append failed";
```

```
"Alert" = "Alert";
"Available" = "Available";
"Cancel" = "Cancel";
"Change Pane" = "Change Pane";
"Clear" = "Clear";
"Close" = "Close";
```

Then, look in the `German.lproj` directory in the same framework. Its string table begins with these string pairs:

```
"About Web Objects" = "Kurzinformation";
"Actions" = "Aktionen";
"Activate Previous Window" = "Fenster wechseln";
"Add" = "Anfügen";
"Add failed" = "Anfügen fehlgeschlagen";
"Append" = "Anhängen";
"Append failed" = "Anhängen fehlgeschlagen";
"Alert" = "Achtung";
"Available" = "Verfügbar";
"Cancel" = "Abbrechen";
"Change Pane" = "Ansicht wechseln";
"Clear" = "Leeren";
"Close" = "Schließen";
```

You can see that the strings are localized for German. Simply copy the string pairs you want to provide localization for into your `Localizable.strings` tables and localize them accordingly.

Building Custom List Controllers

The public methods provided by the controller factory (`com.webobjects.eogeneration.EOControllerFactory`) allow you to dynamically generate user interfaces for many types of tasks throughout your application. However, it doesn't provide methods for all types of tasks, such as list controllers. This topic describes how to programmatically create a list controller.

Problem: You want to display a list controller containing the enterprise objects returned by a fetch.

Solution: Programmatically create a list controller.

The following method constructs a list controller by first creating a generic controller, then by asking the controller factory for a list controller based on the generic controller and an entity name, and then by invoking `listObjectsWithFetchSpecification` to fetch enterprise objects into the list controller.

```
public void listWithEntityName(String entityName, EOFetchSpecification fs) {
    EOControllerFactory f = EOControllerFactory.sharedControllerFactory();

    EOController controller = f.controllerWithSpecification(new NSDictionary (new
       Object[] {entityName, EOControllerFactory.ListTask,
        EOControllerFactory.TopLevelWindowQuestion}, new Object[]
       {EOControllerFactory.EntitySpecification, EOControllerFactory.TaskSpecification,
        EOControllerFactory.QuestionSpecification}), true);

    if (controller != null) {
        EOListController listController =
(EOListController)f.controllerWithEntityName(controller,
           EOControllerFactory.List.class, entityName);
    listController.listObjectsWithFetchSpecification(fs);
    listController.setEditability(EOEditable.NeverEditable);
    listController.makeVisible();
    }
}
```

Using HTML on the Client

The Swing toolkit usually provides all the user interface elements you need for desktop applications. However, there are circumstances in which you want to display HTML in the client application. Fortunately, Swing provides an HTML editing kit object that parses and displays HTML.

You can use WebObjects components and dynamic elements on the server to generate HTML pages, which you then pass back to the client as a String of HTML markup. This gives you the best of both worlds: a rich Swing user interface with the flexibility and power of WebObjects dynamically generated HTML pages.

One limitation of a Swing-based user interface is that it doesn't give you the look and feel flexibility that HTML-based user interfaces do. By providing a small number of HTML components in Java Client applications, you can provide your customers with an application more tailored to their company or their specific needs.

This chapter leads you through this process in the context of providing user help files in a Direct to Java Client application. It is divided into the following sections:

- "Build the HTML Templates" (page 245)
- "Write a Controller Class" (page 246)
- "Add a Menu Item" (page 247)
- "Generate the HTML" (page 247)
- "Retrieve HTML From the Server " (page 248)

Build the HTML Templates

To use HTML on the client, you need to first write a WOComponent that provides a template for the HTML. The document *WebObjects Web Applications Programming Guide* describes how to do this in depth. This book also provides a short tutorial on building WOComponents using WebObjects Builder. See "Write the Action (Build a WOComponent)" (page 122).

The HTML you'll provide to the client in this example is used to display help information for the client application when the user chooses the Help item from the Help menu (you'll add this in "Add a Menu Item" (page 247)). So, the WOComponent need only contain static text that provides usage information.

However, you are free to use any of the WebObjects dynamic elements in the WOComponent. But keep in mind that some of the dynamic elements may generate HTML that is not supported by the Swing HTML editing kit you'll use.

The remainder of this chapter assumes that you create a WOComponent named `HelpWindow.wo`.

Write a Controller Class

Since Direct to Java Client does not provide a controller for an HTML editing kit, you have to write one. The HTML controller is simply another kind of widget, so the custom controller class you write can inherit from EOWidgetController. The HTML editing kit object you'll use is an instance of `javax.swing.text.html.HTMLEditorKit`.

Listing 23-1 shows the HelpWindowController class that the JCRealEstatePhotos example uses.

Listing 23-1 Custom controller class for an HTML editing kit object

```
package webobjectsexamples.realestatephotos.client;

import java.awt.*;
import javax.swing.*;
import javax.swing.text.*;
import javax.swing.text.html.*;
import com.webobjects.eocontrol.*;
import com.webobjects.eogeneration.*;
import com.webobjects.eoapplication.*;
import com.webobjects.eodistribution.client.*;

public class HelpWindowController extends EOWidgetController {

    public HelpWindowController(EOXMLUnarchiver unarchiver) {
        super(unarchiver);
    }

    protected JComponent newWidget() {
        JEditorPane editorPane;
        JScrollPane scrollPane;

        editorPane = new JEditorPane();
        editorPane.setEditable(false);
        editorPane.setEditorKit(new HTMLEditorKit());
        editorPane.setText(helpText());
        editorPane.setPreferredSize(new Dimension(200, 400));
       scrollPane = new JScrollPane(editorPane, JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
            JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);

        return scrollPane;
    }
}
```

The HTMLEditorKit object is placed in an JEditorPane, which in turn is placed in a JScrollPane. The content of the HTMLEditorKit is set to the return value of the `helpText` method, which you'll write in "Retrieve HTML From the Server " (page 248).

To use this controller class, you need to add a D2WComponent to your application which contains an XML description of a controller hierarchy of which the custom controller is a part. To learn how to add a D2WComponent, see "New D2WComponent" (page 186). Name the new component HelpWindowComponent. The XML description, assuming that the controller class that includes the scroll view is named HelpWindowController, could be as shown in Listing 23-2.

Listing 23-2 HTML controller in XML description

```
<FRAMECONTROLLER reuseMode="never" horizontallyResizable="true"
 verticallyResizable="true" label="Search Results" disposeIfDeactivated="true"
 minimumHeight="300" minimumWidth="400">
    <CONTROLLER
         className="webobjectsexamples.realestatephotos.client.HelpWindowController"/>
</FRAMECONTROLLER>
```

To display this controller, you need to register a new task with the rule system. You can use this rule:

Left-Hand Side:	`task = 'help'`
Key:	`window`
Value:	`"HelpWindowComponent"`
Priority:	`50`

This rule is fired when the user selects the Help item from the Help menu, which you'll add in "Add a Menu Item" (page 247).

Add a Menu Item

The user help is accessed from a new menu item called Help, which is in the Help menu. As described in "Adding Custom Menu Items" (page 185), you add menu items to Direct to Java Client applications by adding to the application's `actions` or `additionalActions`. Since the menu item you'll add will be part of the Help menu, you want to add a menu-specific action as described in "Menu-Specific Actions" (page 187).

To add a Help menu item to the Help menu that invokes a task in the rule system called `help`, add the XML shown in Listing 23-3 to a D2WComponent that describes an application's actions. This rule system looks for actions and additional actions in a D2WComponent, which is specified as the right-hand side value of a rule that has a left-hand side of `actions` (or `additionalActions`). The JCRealEstatePhotos example identifies the application's actions in the UserActions component.

Listing 23-3 Specify a Help menu action

```
<ARRAY>
<HELPWINDOWACTION task="help" menuAccelerator="shift T"descriptionPath="Help"/>
</ARRAY>
```

Selecting the Help item from the Help menu invokes the `help` task in the rule system which displays the HelpWindowComponent that includes the HelpWindowController class.

Generate the HTML

The HTML controller class you wrote retrieves its data from the WebObjects application on the server. This means that you can leverage the full power of WebObjects component-driven HTML generation, including most of the dynamic elements and the full power of Enterprise Objects and provide the results in a Swing user interface on the client.

All the HTML controller class needs for input is a string of HTML markup. You need to perform the following operations on the WebObjects application server to provide this HTML markup:

- ask the application instance to generate a WOComponent
- get the content of that component
- convert that content into a string

You can use the code in Listing 23-4 to do this.

Listing 23-4 Generate a WOComponent and convert it to a string of markup

```
public String clientSideRequestGenerateHelp() {
        NSData data;
        WOResponse response;
        WOComponent page;

        page = WOApplication.application().pageWithName("HelpWindow",
   context());

        response = page.generateResponse();
        data = response.content();
        return new String(data.bytes(0, data.length()));
 }
```

Retrieve HTML From the Server

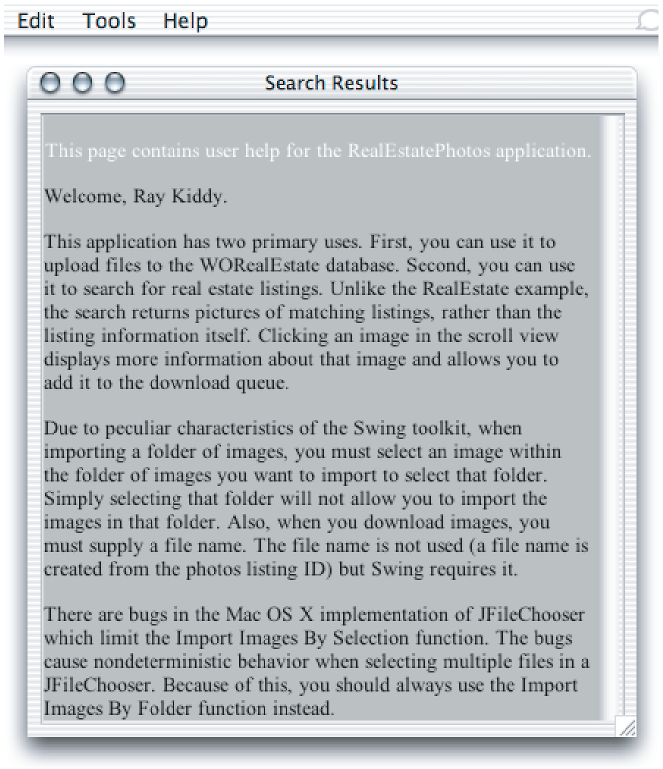
To invoke the method you wrote in "Generate the HTML" (page 247), you need to perform a remote method invocation from the client application. In "Write a Controller Class" (page 246), the content of the editor pane that contains the HTML editing kit is set to the return value of the `helpText` method. That method performs a remote method invocation, which invokes the method on the server side that generates the HTML and returns it as a string. You can use the code in Listing 23-5 to do this.

Listing 23-5 Perform a remote method invocation to generate and return HTML

```
public String helpText() {
        return
(String)_distributedObjectStore().invokeStatelessRemoteMethodWithKeyPath(
  "session", "clientSideRequestGenerateHelp", null, null);
    }

    private EODistributedObjectStore _distributedObjectStore() {
        EOObjectStore objectStore = EOEditingContext.defaultParentObjectStore();
       if ((objectStore == null) || (!(objectStore instanceof EODistributedObjectStore)))
 {
            throw new IllegalStateException("Default parent object store needs to be an
                 EODistributedObjectStore");
        }
        return (EODistributedObjectStore)objectStore;
    }
```

When you put all the ideas in this chapter together, you should see a new item in the Help menu called Help. Selecting this should display the HelpWindowComponent which contains the HTML view and the HTML that is generated by the server-side application. Figure 23-1 shows an example.

Figure 23-1 HTML help in the client



This page contains user help for the RealEstatePhotos application.

Welcome, Ray Kiddy.

This application has two primary uses. First, you can use it to upload files to the WORealEstate database. Second, you can use it to search for real estate listings. Unlike the RealEstate example, the search returns pictures of matching listings, rather than the listing information itself. Clicking an image in the scroll view displays more information about that image and allows you to add it to the download queue.

Due to peculiar characteristics of the Swing toolkit, when importing a folder of images, you must select an image within the folder of images you want to import to select that folder. Simply selecting that folder will not allow you to import the images in that folder. Also, when you download images, you must supply a file name. The file name is not used (a file name is created from the photos listing ID) but Swing requires it.

There are bugs in the Mac OS X implementation of JFileChooser which limit the Import Images By Selection function. The bugs cause nondeterministic behavior when selecting multiple files in a JFileChooser. Because of this, you should always use the Import Images By Folder function instead.

Building a Login Window

Most of the tasks you need to perform to implement a login window in a Direct to Java Client application are explained in earlier task chapters. This task provides a road map and some helpful hints to successfully implement a login window in your application.

The WebObjects Java Client examples use the JCAuthentication framework to provide a sophisticated login window for Java Client applications that includes logout, password change, and registration capabilities. It is considerably more complex than the simple login window this chapter describes and you'll probably want to use it to implement authentication functionality in your applications.

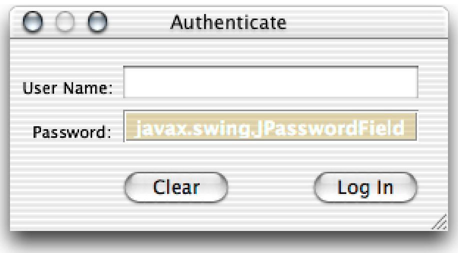
However, the JCAuthentication framework is rather complex and is difficult to understand for those new to Java Client development. This chapter is provided to help you understand the basic concepts in implementing access controls in a Java Client application. After you grasp the concepts in this chapter, you should seriously consider using the JCAuthentication framework in your application rather than an authentication framework you build from start to finish without the benefit of any pre-built components.

First, you'll build the login window's user interface. Then, you'll provide logic for authenticating users to a data store. Finally, you'll intercept the default startup sequence for Direct to Java Client applications to disable certain menu items and windows.

Building the User Interface

The first step in building a login window is to build the user interface for it. Add a nib file to your project and open it in Interface Builder. Add two text fields with labels and two buttons. A suggestion appears in Figure 24-1.

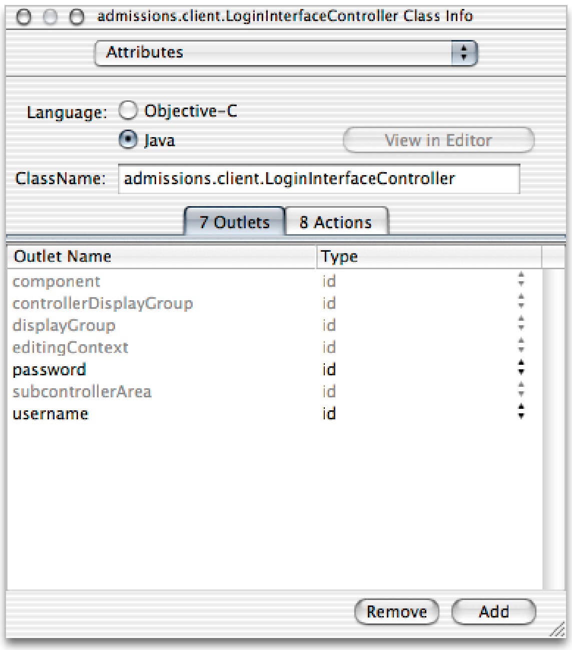
Figure 24-1 Login window user interface



If you want to make the password text field secure (so that typing in it produces asterisks rather than characters), see "Custom Views" (page 213) to learn how to add custom view widgets to a nib file. Substitute `javax.swing.JPasswordField` in place of the custom view widget used in that section.

Then, add outlets to File's Owner for each text field, naming them `username` and `password`. See "Custom Views" (page 213) or "Programmatic Access to Interface Components" (page 151) to learn how to add outlets. The outlets pane should then appear as shown in Figure 24-2.

Figure 24-2 Add outlets named `username` and `password`



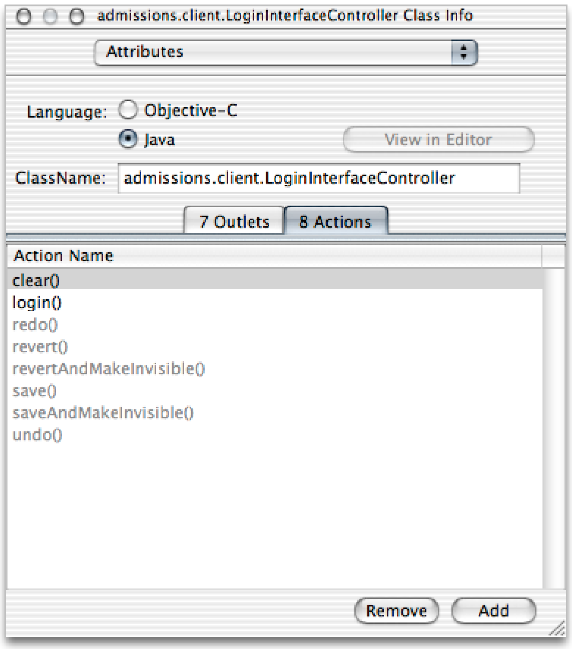
Language: ◯ Objective-C [✓ ● Java]

ClassName: admissions.client.LoginInterfaceController

Outlet Name	Type
component	id
controllerDisplayGroup	id
displayGroup	id
editingContext	id
password	id
subcontrollerArea	id
username	id

Also add two new actions called `login` and `clear`. You add actions the same way as you add outlets except you add them in the Actions pane rather than in the Outlets pane. The Actions pane should then appear as in Figure 24-3.

Figure 24-3 Add actions


admissions.client.LoginInterfaceController Class Info

Attributes

Language: Objective-C [✓ Java]

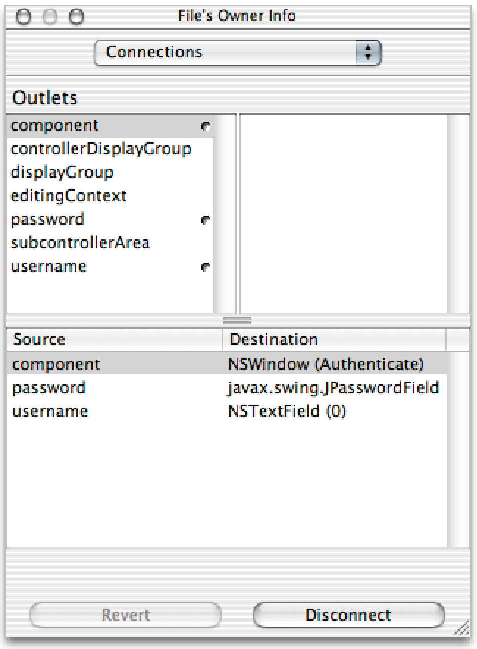
ClassName: admissions.client.LoginInterfaceController

7 Outlets | 8 Actions

Action Name
clear()
login()
redo()
revert()
revertAndMakeInvisible()
save()
saveAndMakeInvisible()
undo()

Finally, connect the outlets and the actions to the widgets you added. Control-drag from File's Owner to the User Name text field and bind it to the `username` outlet. Control-drag from File's Owner to the Password text field and bind it to the `password` outlet. Control-drag from the Log In button to File's Owner and bind its target aspect to the `login` action. Control-drag from the Clear button to File's Owner and bind its target aspect to the `clear` action. Also make sure that File's Owner `component` outlet is bound to the nib file's Window object. The File's Owner connections should then appear as shown in Figure 24-4.

Figure 24-4 File's Owner with new connections

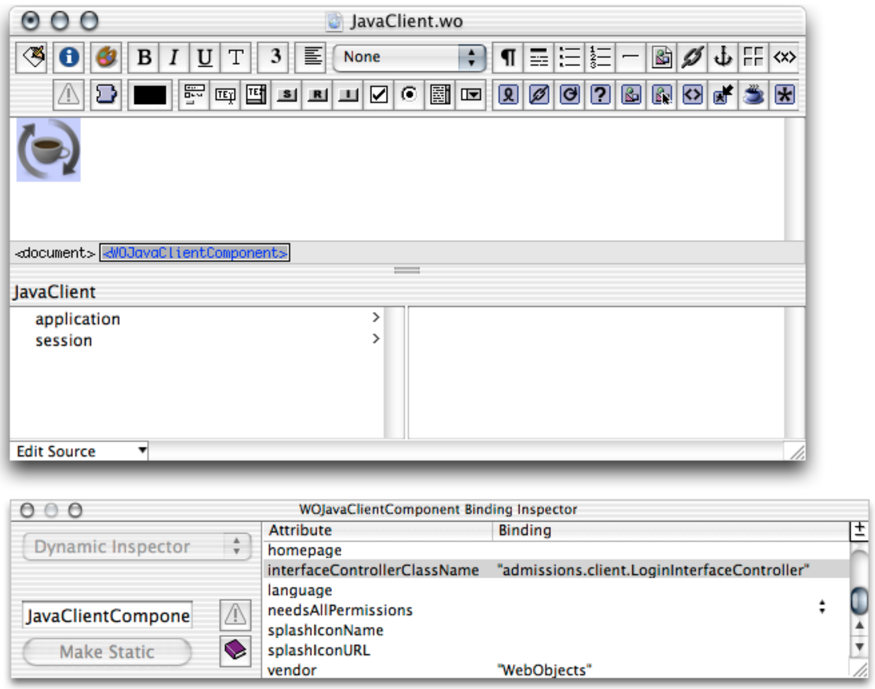


Source	Destination
component	NSWindow (Authenticate)
password	javax.swing.JPasswordField
username	NSTextField (0)

Save the nib file.

To load this interface when the application launches, you use a binding provided on the JavaClient component. Open `JavaClient.wo` in WebObjects Builder and select the WOJavaClientComponent dynamic element, as shown in Figure 24-5.

Figure 24-5 Select the WOJavaClientComponent dynamic element



Attribute	Binding
homepage	
interfaceControllerClassName	"admissions.client.LoginInterfaceController"
language	
needsAllPermissions	
splashIconName	
splashIconURL	
vendor	"WebObjects"

Then, open the WOJavaClientComponent Binding Inspector by choosing Inspector from the Window menu. This shows you all the possible bindings for the WOJavaClientComponent dynamic element. For the `interfaceControllerClassName` binding, enter the fully qualified name of the login nib file you created in this chapter, making sure to put it in quotation marks. An example appears in Figure 24-5.

Save the JavaClientcomponent.

The nib file specified as the value for this binding is loaded when the application starts up. Before you can use the nib file, you need to add instance variables in its controller class that correspond to the outlets you added to the nib. You'll do this in the next section.

Adding Logic to Authenticate Users

Now that you have a user interface for the login window, you need to add logic to authenticate users. The first step is to add an entity called Person to your application's EOModel and generate a table for it in the database. This is the table against which your users will be authenticated. The entity needs three attributes: a primary key, an attribute named "username," and an attribute named "password." (The sample code in this section assumes these names but there is no reason you can't use an entity named "User" and attributes with different names).

You also need to generate client and server Java class files for the Person entity. See "Add Custom Business Logic" (page 106) for a reminder of how to do this using EOModeler.

The second step is to extract the values of the two text fields in the nib file. To do this, you need access to the text fields in the nib file's controller class, as described in "Programmatic Access to Interface Components" (page 151). Add an instance variable of type EOTextField for both of the text fields in the nib file. The instance variable's names must correspond to the name of the outlets with which the text fields are connected. If you made the password field a JPasswordField, make its type JPasswordField.

Listing 24-1 Outlets using plain-text fields

```
public EOTextField username, password;
```

Listing 24-2 Outlets using one password field

```
public EOTextField username;
public JPasswordField password;
```

EOTextField is defined in the EOInterface Swing package and JPasswordField in the standard Swing package, so add import statements for both packages:

```
import com.webobjects.eointerface.swing.*;
import javax.swing.*;
```

The client part of the application needs a reference to an authenticated user. Since you added a Person entity to your model, a user is an object of type Person. Add another instance variable for a user:

```
private Person _user;
```

Now, add methods for the actions you added to the nib file. You added two actions, `clear` and `login`, so add two methods with those names to the nib file's controller class:

```
public void login() {}
public void clear() {}
```

The `clear` method simply clears the values of the text fields. Add this code to the class:

```
username.setText("");
password.setText("");
```

The `login` method authenticates users by sending the user-entered values from the User Name and Password text fields to remote methods on the server-side application, which query a data store to perform the authentication. If a user successfully authenticates, the client-side method that invoked the server-side method receives an object (an EOGlobalID) representing the user who authenticated.

Add the method in Listing 24-3 (page 256) to the nib file's controller class to perform the remote method invocation. If the user successfully authenticates, the method returns `true` and sets the `_user` instance variable to the enterprise object representing the user who successfully authenticated.

Listing 24-3 Client-side login method

```
public boolean clientSideRequestLogin() {
    EOGlobalID person =
         (EOGlobalID)(_distributedObjectStore().invokeStatelessRemoteMethodWithKeyPath(
         "session", "clientSideRequestLogin", new Class[] {String.class, String.class},
           new Object[] {username.getText(), new String(password.getPassword())}));
    if (person != null) {
        EOEditingContext ec = new EOEditingContext();
        _user = (Person)(ec.faultForGlobalID(person, ec));
        return true;
    }
    else
        return false;
}
```

Remember to also add the method that returns the client's parent object store, as described in "Distributed Object Store" (page 97) since the remote method invocation is invoked on the client's parent object store. Remember to also add the import statement for the distributed object store's package:

```
import com.webobjects.eodistribution.client.*;
```

Now, invoke the client-side method `clientSideRequestLogin` in the `login` method, adding a conditional based on the response, as shown in Listing 24-4.

Listing 24-4 The `login` method

```
public void login() {
     if (this.clientSideRequestLogin()) {
        //allow user into application
     }
     else {
        EODialogs.runErrorDialog("Login failed", "Login failed. Please try
           again.");
        }
    }
```

This is all you need to do on the client side. Now, you need to add the method on the server-side that actually performs the authentication. The remote method invocation specifies the key path `session` and the method `clientSideRequestLogin`, so add a method in `Session.java` with that name, as shown in Listing 24-5 (page 257). The `clientSideRequestLogin` method uses the EOUtilities class which is defined in the access layer, so also add the import statement for `com.webobjects.eoaccess` in `Session.java`.

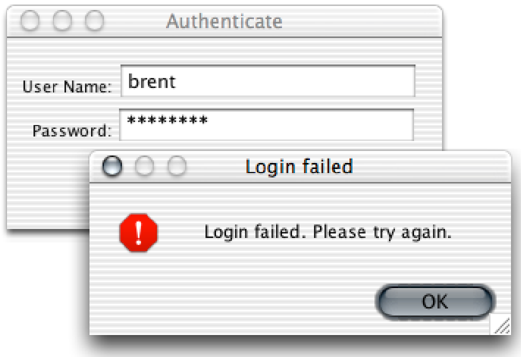
Listing 24-5 Authentication in `Session.java`

```
public EOGlobalID clientSideRequestLogin(String username, String password) {
        EOGenericRecord user;
        EOEditingContext editingContext = new EOEditingContext();

        NSMutableDictionary userCredentials = new NSMutableDictionary();
        userCredentials.setObjectForKey(username, "username");
        userCredentials.setObjectForKey(password, "password");

       NSArray foundObjects = EOUtilities.objectsMatchingValues(editingContext, "Person",
         userCredentials);
        if (foundObjects.count() == 1) {

            user = (EOGenericRecord)foundObjects.objectAtIndex(0);

            return(editingContext.globalIDForObject(user));
        }
        else {
            return null;
        }
    }
```

This method constructs a dictionary based on the values passed in from the client side (the user-entered name and password). Then, using the class `com.webobjects.eoaccess.EOUtilities`, the method performs a fetch against the data store in the Person entity. If a record matching the user's credentials is found, the method returns the EOGlobalID for that user.

The client-side method `clientSideRequestLogin` receives the result of this method, and if it is not `null`, allows the user into the application. If it receives `null`, however, it displays a dialog with an error message, as shown in Figure 24-6.

Figure 24-6 Login failed



Login failed. Please try again.

Of course, authentication fails if you don't add users to the entity in the data store on which you perform the fetch specification, so remember to add users.

Restricting Access

The login window you added won't be of much use until you change the default startup sequence to remove the Documents menu and the default query window. Otherwise, users can simply ignore the login window and start using the application.

To learn how to remove the Documents menu, see "The Documents Menu" (page 177). To learn how to suppress the default query window, see "The Default Query Window" (page 178).

Finally, now that you've disabled all the default mechanisms for users to use the application, you need to provide custom access. In Listing 24-4 (page 257), currently nothing happens if the user successfully authenticates—except that they don't see the error dialog stating that authentication failed.

However, there are many things you can do, such as using the controller factory programmatically ("Generating Controllers With the Controller Factory" (page 181)) or loading another nib file that provides a menu of the application's primary tasks. You can display the default query window after the user authenticates by adding this code in the `if` part of the conditional in the `login` method in Listing 24-4 (page 257):

```
EOControllerFactory.sharedControllerFactory().queryControllerWithEntity
("entityName")
```

Make sure to add the import statement for the `com.webobjects.eogeneration` package to the nib file.

To load a nib file programmatically, change the `login` method in Listing 24-4 (page 257) to Listing 24-6 (page 259).

Listing 24-6 Load a nib file programmatically

```
public void login() {
     if (this.clientSideRequestLogin()) {
        MainMenuInterfaceController mainMenu = new                    // 1
             MainMenuInterfaceController();                           // 2
        EOFrameController.runControllerInNewFrame(mainMenu, null);    // 3
     }
     else {
        EODialogs.runErrorDialog("Login failed", "Login failed. Please try
      again.");
        }
}
```

Code line 1 instantiates a new instance of the nib file named `MainMenuInterfaceController` and code line 2 displays the nib file.

Document Revision History

This table describes the changes to *WebObjects Java Client Programming Guide*.

Date	Notes
2005-08-11	Changed the title from "Java Client Desktop Applications."
2002-11-01	Updated for WebObjects 5.2.
	Added information on layout hints and levels enhancements to the dynamic user-interface generation, "Widgets Pane" (page 90)
	Added information on the new security contract for remote method invocations in the distribution layer, "Business Logic" (page 95) and "Delegates" (page 100)
	Added information on how to implement SSL in the distribution layer, "Using SSL" (page 99)
	Added information about delegates in the distribution layer, "Delegates" (page 100), and how to set them "Setting the Delegate" (page 101)
	Added information on how to deploy the client application, "Deploying Client Applications" (page 169)
	Added information on new controllers and new arguments, "Controllers and Actions Reference" (page 265)
	Added information on how to package client applications as Mac OS X desktop applications, "Desktop Applications" (page 173)
	Added information on how to implement the new continuous change notification feature in certain controllers, "Continuous Change Notification" (page 197)
	Added information on how to build the WOComponent in "Write the Action (Build a WOComponent)" (page 122)
	Added "Building Custom Controllers With XML" (page 155)
	Added "Using HTML on the Client" (page 245)
	Added "Development Process Overview" (page 29)
	Added "Using Nib Actions When Mixing " (page 211)
	Separated information on the Direct to Java Client Assistant into its own chapter, "Inside Assistant" (page 85)
	Adapted task chapters more closely to the JCRealEstatePhotos example

Date	Notes
	Changed the title of the chapter "Using the Controller Factory Programmatically" to "Generating Controllers With the Controller Factory" (page 181)
	Removed "Task:" in the title of the task chapters
	Updated "Controllers and Actions Reference" (page 265) for WebObjects 5.2
	Moved information on different deployment strategies from the introduction to "Deploying Client Applications" (page 169)
	D2WContext key `languages` is now `locales`, "Localizing Property Labels" (page 235)
	The security contract in the distribution layer changed, "Business Logic" (page 95) and "Delegates" (page 100)
	The package `com.webobjects.eogeneration.client` changed to `com.webobjects.eogeneration`
	Web Start is now integrated with Java Client, "Web Start" (page 170) and "Server Files (Application Server Target)" (page 65)
	Removed most information about deploying as applets, as applet support is now deprecated
	It's no longer necessary to set the class of File's Owner in nib files, "Prepare the Nib File" (page 141)
	MouseInputAdapter class is now correctly named, "Make EOImageView Accept Clicks" (page 218)
2002-05-01	Updated for WWDC.
	Added "Restricting Access to an Application" (page 177)
	Added "Generating Controllers With the Controller Factory" (page 181)
	Added "Adding Custom Menu Items" (page 185)
	Added "Adding Custom Actions to Controllers" (page 189)
	Added "Common Rules" (page 193)
	Added "Freezing XML User Interfaces" (page 199)
	Added "Mixing Static and Dynamic User Interfaces" (page 209)
	Added "Using Custom Views in Nib Files" (page 213)
	Added "Localizing Dynamic Components" (page 235)
	Added "Building Custom List Controllers" (page 243)
	Added "Using and Extending Image Views in Nib Files" (page 221)

Date	Notes
	Added “Using Pop-up Menus in Nib Files” (page 227)
	Added “Building a Login Window” (page 251)

Controllers and Actions Reference

This appendix provides an overview of the classes used in Java Client applications, including the XML descriptions, tags, and attributes they use. Refer to "XML Value Types" (page 265) for complete information on the proprietary value types such as `editability` and `alignment`.

XML Value Types

The XML attributes for Java Client classes include standard Java value types and these other value types:

position

`Center`

`Top`

`Bottom`

`Left`

`Right`

`TopLeft`

`TopRight`

`BottomLeft`

`BottomRight`

border

`None`

`Etched`

`RaisedBezel`

`LoweredBezel`

`LineBorder`

editability

`Never`

`Always`

`IfSuperController`

alignment

`Center`

`Left`

`Right`

insets

The format for this value type is " *`top`*, *`left`*, *`bottom`*, *`right`*", with each coordinate specified by an integer.

resizing

`NoResizing`

`AspectResizing`

`FreeResizing`

`IntegralResizing`

`PerformanceResizing`

`HorizontalResizing`

`VerticalResizing`

scaling

`ScaleNone`

`ScaleProportionally`

`ScaleToFit`

`ScaleProportionallyIfTooLarge`

scaling hint

`ScaleDefault`

`ScaleFast`

`ScaleSmooth`

color

Specify by three integers each in the range 0 to 255: "*integerRed*, *integerGreen*, *integerBlue*", or specify by hex: "#FFFFFF".

font

`Size`

`Style`

`Font`

The format for this value type is "*size*, *style*:*fontName*". Specify `Size` as "+ *integer*" or "-*integer*". Specify `Style` as `Plain`, `Bold`, `Italic`, or `BoldItalic`.

Example: `<CONTROLLER className="com.myapp.TextFieldController" font="+2, Bold: Arial"/>`

provider method

`ClassName`

`MethodName`

The format for this value type is "*className*: *methodName*".

Controllers With XML Tags and XML Attributes

Direct to Java Client user interfaces are defined in XML descriptions. The XML descriptions identify controllers, which in turn represent Enterprise Objects objects and Swing user interface objects. This section lists all the controllers in the Java Client frameworks that have at least one XML tag or one XML attribute. Controllers inherit XML attributes, but inherited XML attributes may not always be appropriate for the inheriting controller.

The following list is alphabetical by controller name, without regard to a controller's package.

com.webobjects.eoapplication.EOActionButtonsController

Superclass:

`com.webobjects.eoapplication.EOActionWidgetController`

Description:

Handles toolbars with multiple action buttons.

XML Tag:

`ACTIONBUTTONSCONTROLLER`

XML Attributes:

`usesLargeButtonRepresentation` (`boolean`)

com.webobjects.eogeneration.EOActionController

Superclass:

`com.webobjects.eogeneration.EOTitlesController`

Description:

Handles invoking actions and action keys.

XML Tag:

`ACTIONCONTROLLER`

XML Attributes:

`actionKey` (`String` representing the key path for the action aspect of the association)

`buttonPosition` (position)

`usesAction` (`boolean`)

`usesButton` (`boolean`)

com.webobjects.eoapplication.EOActionMenuController

Superclass:

`com.webobjects.eoapplication.EOActionWidgetController`

Description:

Handles pop-up menus with multiple action items.

XML Tag:

`ACTIONMENUCONTROLLER`

com.webobjects.eoapplication.EOActionTrigger

Superclass:

`com.webobjects.eoapplication.EOComponentController`

Description:

A lightweight control for actions.

XML Tag:

`ACTIONTRIGGER`

XML Attributes:

`usesLargeButtonRepresentation` (`boolean`)

com.webobjects.eoapplication.EOActionWidgetController

Superclass:

`com.webobjects.eoapplication.EOComponentController`

Description:

Handles toolbars and other action controls.

XML Tag:

None (abstract class)

XML Attributes:

`widgetPosition` (position)

com.webobjects.eoapplication.EOArchiveController

Superclass:

`com.webobjects.eoapplication.EOComponentController`

Description:

Used for loading interface files.

XML Tag:

None (abstract class)

XML Attributes:

`archive` (`String` representing the name of the interface file to be used by a controller rather than the dynamic user-interface generation)

com.webobjects.eogeneration.EOAssociationController

Superclass:

`com.webobjects.eogeneration.EOWidgetController`

Description:

Handles associations for widgets, including the editable state and enabled key.

XML Tag:

None (abstract class)

XML Attributes:

`displayGroupProviderMethodName` (provider method name)

`editability` (editability)

`enabledDisplayGroupProviderMethodName` (provider method name)

`enabledKey` (`String` representing the key path for the enabled aspect of the association)

`prefersContinuousChangeNotification` (`boolean`)

`suppressesAssociation` (`boolean`)

com.webobjects.eoapplication.EOBoxController

Superclass:

`com.webobjects.eoapplication.EOComponentController`

Description:

Displays bordered and titled boxes.

XML Tag:

`BOXCONTROLLER`

XML Attributes:

`borderType` (border)

`color` (color)

`font` (font)

`highlight` (`boolean`)

`titlePosition` (position)

`usesTitledBorder` (`boolean`)

com.webobjects.eogeneration.EOCheckBoxController

Superclass:

`com.webobjects.eogeneration.EOValueController`

Description:

Handles checkboxes.

XML Tag:

`CHECKBOXCONTROLLER`

XML Attributes:

`displaysLabelInWidget` (boolean)

com.webobjects.eogeneration.EOComboBoxController

Superclass:

`com.webobjects.eogeneration.EOTitlesController`

Description:

Handles combo boxes with fixed values.

XML Tag:

`COMBOBOXCONTROLLER`

XML Attributes:

`isQueryWidget` (boolean)

`valueKey` (`String` representing the key path for the value aspect of the association)

com.webobjects.eoapplication.EOComponentController

Superclass:

`com.webobjects.eoapplication.EOController`

Description:

Handles user interface issues such as visibility state, labels, icons, size information, subcontroller layout; default component controller is an empty box.

XML Tag:

`COMPONENTCONTROLLER`

XML Attributes:

`alignsComponents` (boolean)

`horizontallyResizable` (boolean)

`iconName` (String)

`iconURL` (String)

`insets` (insets)

`label` (String)

`minimumHeight` (int)

`minimumWidth` (int)

`prefersIconOnly` (boolean)

`toolTip` (String)

`usesHorizontalLayout` (boolean)

`verticallyResizable` (boolean)

com.webobjects.eoapplication.EOController

Superclass:

`java.lang.Object`

Description:

An abstract definition of controllers; handles supercontrollers, subcontrollers, key-value coding for the controller hierarchy, messages in the controller hierarchy, establishing and breaking connections to other controllers.

XML Tag:

None (abstract class)

XML Attributes:

`actions` (array of EOAction objects)

`className` (String)

`disabledActionNames` (array of strings with names of actions to be disabled)

`keyValuePairs` (additional values that can be looked up with key-value coding)

`transient` (boolean)

`typeName` (String)

com.webobjects.eogeneration.EODefaultActionTrigger

Superclass:

`com.webobjects.eogeneration.EOAssociationController`

Description:

None (abstract class)

XML Tag:

`invokesDefaultAction` (boolean)

com.webobjects.eogeneration.EODetailSelectionController

Superclass:

`com.webobjects.eogeneration.EOEnumerationController`

Description:

Handles detail selection tables.

XML Tag:

`DETAILSELECTIONCONTROLLER`

XML Attributes:

None

com.webobjects.eoapplication.EODialogController

Superclass:

`com.webobjects.eoapplication.EOSimpleWindowController`

Description:

Handles dialogs (such as error messages).

XML Tag:

`DIALOGCONTROLLER`

com.webobjects.eogeneration.EODisplayStatisticsController

Superclass:

`com.webobjects.eogeneration.EOStaticLabelController`

Description:

Displays the number of selected objects and the number of displayed objects in list controllers.

XML Tag:

`DISPLAYSTATISTICSCONTROLLER`

XML Attributes:

`displayGroupProviderMethodName` (provider method)

`displayPattern` (`String`)

com.webobjects.eoapplication.EODocumentController

Superclass:

`com.webobjects.eoapplication.EOEntityController`

Description:

Handles editable documents.

XML Tag:

DOCUMENTCONTROLLER

XML Attributes:

`editability` (editability)

com.webobjects.eogeneration.EOEditingController

Superclass:

`com.webobjects.eoapplication.EODocumentController`

Description:

Handles master-detail associations.

XML Tag:

None

XML Attributes:

`path` (`String`, representing a relationship key path)

`mandatoryRelationshipPaths` (array of `String` values, representing the relationships to be filled on insert)

com.webobjects.eoapplication.EOEntityController

Superclass:

`com.webobjects.eoapplication.EOComponentController`

Description:

Handles business data on the level of entities: entity names, editing context, display group, controller display group, loading archives (Interface Builder files).

XML Tag:

ENTITYCONTROLLER

XML Attributes:

`entity` (`String` representing the entity)

`displayGroupProviderMethodName` (provider method)

`editingContextProviderMethodName` (provider method)

`fetchesOnConnectEnabled` (`boolean`)

com.webobjects.eogeneration.EOEnumerationController

Superclass:

`com.webobjects.eogeneration.EOTitlesController`

Description:

Handles enumeration widgets.

XML Tag:

None (abstract class)

XML Attributes:

`path` (`String` representing the detail relationship path to title objects)

com.webobjects.eogeneration.EOFormatValueController

Superclass:

`com.webobjects.eogeneration.EOValueController`

Description:

Handles the formatting and value aspect of widgets with associations.

XML Tag:

None (abstract class)

XML Attributes:

`formatAllowed` (`boolean`)

`formatClass` (`String` of the class name of the formatter object)

`formatPattern` (`String` representing the pattern string of the formatter object)

com.webobjects.eogeneration.EOFormController

Superclass:

`com.webobjects.eogeneration.EOEditingController`

Description:

Handles editable forms.

XML Tag:

`FORMCONTROLLER`

com.webobjects.eoapplication.EOFrameController

Superclass:

`com.webobjects.eoapplication.EOSimpleWindowController`

Description:

Handles frames.

XML Tag:

`FRAMECONTROLLER`

com.webobjects.eogeneration.EOImageViewController

Superclass:

`com.webobjects.eogeneration. EOValueAndURLController`

Description:

Handles image views.

XML Tag:

`IMAGEVIEWCONTROLLER`

XML Attributes:

`imageScaling` (scaling)

`scalingHints` (scaling hint)

com.webobjects.eoapplication.EOInspectorController

Superclass:

`com.webobjects.eoapplication.EOWindowController`

Description:

Handles inspector windows.

XML Tag:

`INSPECTORCONTROLLER`

XML Attributes:

`sharedIdentifier` (`String`)

com.webobjects.eoapplication.EOInterfaceController

Superclass:

`com.webobjects.eoapplication.EODocumentController`

Description:

Handles documents that always use an interface file.

XML Tag:

`INTERFACECONTROLLER`

com.webobjects.eogeneration.EOListController

Superclass:

`com.webobjects.eogeneration.EOEditingController`

Description:

Handles editable lists.

XML Tag:

`LISTCONTROLLER`

com.webobjects.eoapplication.EOMenuSwitchController

Superclass:

`com.webobjects.eoapplication.EOSwitchController`

Description:

Handles switch panes that have a pop-up menu.

XML Tag:

`MENUSWITCHCONTROLLER`

com.webobjects.eoapplication.EOModalDialogController

Superclass:

`com.webobjects.eoapplication.EODialogController`

Description:

Handles modal dialog controllers.

XML Tag:

`MODALDIALOGCONTROLLER`

com.webobjects.eogeneration.EOMultipleValuesEnumerationController

Superclass:

`com.webobjects.eogeneration.EOEnumerationController`

Description:

Handles to-many relationships to enumeration entities.

XML Tag:

`MULTIPLEVALUESENUMERATIONCONTROLLER`

XML Attributes:

`allowsDuplicates` (`boolean`)

`allowsRemoveAll` (`boolean`)

`detailKeys` (an array of `String` values, representing the key paths to be displayed for selected objects)

`detailRelationshipPath` (`String`)

`indexKey` (`String`, representing the key to sort on)

`usesTableLabels` (`boolean`)

com.webobjects.eogeneration.EOOneValueEnumerationController

Superclass:

`com.webobjects.eogeneration.EOEnumerationController`

Description:

Handles to-one relationships to enumeration entities.

XML Tag:

`ONEVALUEENUMERATIONCONTROLLER`

XML Attributes:

`isQueryWidget` (`boolean`)

com.webobjects.eoapplication.EOProgrammaticSwitchController

Superclass:

`com.webobjects.eoapplication.EOSwitchController`

Description:

Handles switch views that can only be changed programmatically.

XML Tag:

`PROGRAMMATICSWITCHCONTROLLER`

com.webobjects.eogeneration.EOQueryController

Superclass:

`com.webobjects.eoapplication.EOEntityController`

Description:

Handles query interfaces.

XML Tag:

`QUERYCONTROLLER`

XML Attributes:

`editability` (editability)

`runsConfirmDialogForEmptyQualifiers` (`boolean`)

com.webobjects.eogeneration.EOQuickTimeViewController

Superclass:

`com.webobjects.eogeneration. EOAssociationController`

Description:

Handles QuickTime views.

XML Tag:

`QUICKTIMEVIEWCONTROLLER`

XML Attributes:

`quickTimeCanvasResizing` (resizing)

`URLKey` (`String` representing the key path for the URL aspect of the association)

com.webobjects.eogeneration.EORangeTextFieldController

Superclass:

`com.webobjects.eogeneration.EORangeValueController`

Description:

Handles range text fields with optional formatters.

XML Tag:

`RANGETEXTFIELDCONTROLLER`

XML Attributes:

`formatAllowed` (`boolean`)

`formatClass` (`String` of the class name of the format object)

`formatPattern` (`String` representing the pattern of the format object)

`isQueryWidget` (`boolean`)

com.webobjects.eogeneration.EORangeValueController

Superclass:

`com.webobjects.eogeneration.EORangeWidgetController`

Description:

Handles the value aspect, enabled key aspect, and editable state of range widgets.

XML Tag:

None (abstract class)

XML Attributes:

`displayGroupProviderMethodName` (method name)

`editability` (editability)

`enabledDisplayGroupProviderMethodName` (method name)

`enabledKey` (`String` representing the key path for the enabled aspect of the association)

`maximumValueKey` (`String` representing the key path for the value aspect of the maximum association)

`minimumValueKey`(`String` representing the key path for the value aspect of the minimum association)

`valueKey` (`String` representing the key path for the value aspect of both the minimum and maximum associations)

`suppressesAssociation` (`boolean`)

com.webobjects.eogeneration.EORangeWidgetController

Superclass:

`com.webobjects.eogeneration.EOWidgetController`

Description:

Handles range widgets—two widgets for the minimum and maximum value of the same value.

XML Tag:

None (abstract class)

com.webobjects.eoapplication.EOSimpleWindowController

Superclass:

`com.webobjects.eoapplication.EOWindowController`

Description:

Handles standalone windows.

XML Tag:

None (abstract class)

XML Attributes:

`disposeIfDeactivated` (`boolean`)

com.webobjects.eoapplication.EOSplitController

Superclass:

`com.webobjects.eoapplication.EOComponentController`

Description:

Handles split views (split & snap) which display exactly two subcontroller views.

XML Tag:

`SPLITCONTROLLER`

XML Attributes:

`allowsOneTouchExpandable` (`boolean`)

`allowsSnapToZero` (`boolean`)

`resizeWeight` (`double` from 0.0 to 1.0)

`usesContinuousLayout` (`boolean`)

com.webobjects.eogeneration.EOStaticIconController

Superclass:

`com.webobjects.eoapplication.EOComponentController`

Description:

Handles the display of static icons.

XML Tag:

`STATICICONCONTROLLER`

com.webobjects.eogeneration.EOStaticLabelController

Superclass:

`com.webobjects.eoapplication.EOComponentController`

Description:

Handles the display of static messages.

XML Tag:

`STATICLABELCONTROLLER`

XML Attributes:

`alignment` (alignment)

`color` (color)

`font` (font)

com.webobjects.eogeneration.EOStaticTextFieldController

Superclass:

`com.webobjects.eogeneration.EOTextFieldController`

Description:

Handles uneditable table columns.

XML Tag:

`STATICTEXTFIELDCONTROLLER`

XML Attributes:

`color` (color)

`font` (font)

com.webobjects.eoapplication.EOSwitchController

Superclass:

`com.webobjects.eoapplication.EOComponentController`

Description:

Handles switch views, which only display one view out of many at one time.

XML Tag:

None (abstract class)

com.webobjects.eogeneration.EOTableColumnController

Superclass:

`com.webobjects.eogeneration. EOFormatValueController`

Description:

Handles table columns.

XML Tag:

`TABLECOLUMNCONTROLLER`

com.webobjects.eogeneration.EOTableController

Superclass:

`com.webobjects.eogeneration.EODefaultActionTrigger`

Description:

Handles table views.

XML Tag:

`TABLECONTROLLER`

XML Attributes:

`allowsMultipleSelection` (`boolean`)

`sortsByColumnOrder` (`boolean`)

com.webobjects.eoapplication.EOTabSwitchController

Superclass:

`com.webobjects.eoapplication.EOSwitchController`

Description:

Handles tabbed panes.

XML Tag:

`TABSWITCHCONTROLLER`

com.webobjects.eogeneration.EOTextAreaController

Superclass:

`com.webobjects.eogeneration.EOValueAndURLController`

Description:

Handles scrollable text areas.

XML Tag:

`TEXTAREACONTROLLER`

com.webobjects.eogeneration.EOTextFieldController

Superclass:

`com.webobjects.eogeneration.EOFormatValueController`

Description:

Handles editable text fields.

XML Tag:

`TEXTFIELDCONTROLLER`

XML Attributes:

`isQueryWidget` (boolean)

`usesPasswordField` (boolean)

com.webobjects.eogeneration.EOTitlesController

Superclass:

`com.webobjects.eogeneration.EOAssociationController`

Description:

Handles the attributes of enumeration widgets such as titles, title keys, title display groups, and editable state.

XML Tag:

None (abstract class)

XML Attributes:

`availableTitlesKey` (`String`, representing a key path specifying an array of available title objects)

`resetTitlesObjectsOnEveryConnect` (boolean)

`searchesTitlesObjectsInEditingContext` (boolean)

`titleKeys` (array of `String` values, representing the key paths to be displayed for title objects)

`titlesDisplayGroupProviderMethodName` (method name)

`titlesEntity` (`String`, representing the name of the titles entity; an optional attribute for some subclasses that can derive a name)

com.webobjects.eogeneration.EOTreeController

Superclass:

`com.webobjects.eogeneration.EODefaultActionTrigger`

Description:

Handles tree views.

XML Tag:

`TREECONTROLLER`

XML Attributes:

`allowsDiscontiguousSelection` (boolean)

`allowsMultipleSelection` (boolean)

`childrenKey` (`String`, representing a key path for the `children` aspect of the association)

`expandedIconKey` (`String`, representing a key path for the `expandedIcon` aspect of the association)

`iconKey` (`String`, representing a key path for the `icon` aspect of the association)

`isLeafKey` (`String`, representing a key path for the `isLeaf` aspect of the association)

`isRootVisible` (`boolean`)

`parentKey` (`String`, representing the inverse key path for the `children` key; not used as an aspect)

`rootKey` (`String`, representing the key path for the root aspect of the association)

`valueKey` (`String`, representing the key path for the `value` aspect of the association)

com.webobjects.eogeneration.EOValueAndURLController

Superclass:

`com.webobjects.eogeneration.EOValueController`

Description:

Handles the value or URL aspects of widgets with associations.

XML Tag:

None (abstract class)

XML Attributes:

`URLKey` (`String` representing the key path for the URL aspect of the association)

com.webobjects.eogeneration.EOValueController

Superclass:

`com.webobjects.eogeneration. EOAssociationController`

Description:

Handles the value aspect of widgets with associations.

XML Tag:

None (abstract class)

XML Attributes:

`valueKey` (`String` representing key path for value aspect of association)

com.webobjects.eogeneration.EOWidgetController

Superclass:

`com.webobjects.eoapplication.EOComponentController`

Description:

Handles individual widgets, including their label and alignment.

XML Tag:

None (abstract class)

XML Attributes:

`alignment` (alignment)

`highlight` (`boolean`)

`labelAlignment` (alignment)

`labelComponentPosition` (position)

`labelComponentWidth` (integer)

`minimumWidgetHeight` (integer, which specifies the minimum height just for the widget, not the complete component, which includes the label)

`minimumWidgetWidth` (integer, which specifies the minimum width just for the widget, not the complete component, which includes the label)

`usesLabelComponent` (`boolean`)

com.webobjects.eoapplication.EOWindowController

Superclass:

`com.webobjects.eoapplication.EOComponentController`

Description:

Handles windows and user defaults for window size and position.

XML Tag:

None (abstract class)

XML Attributes:

`windowPosition` (position)

`usesAction` (`boolean`)

`usesButton` (`boolean`)

`usesUserDefaultsWindowLocation` (`boolean`)

`usesUserDefaultsWindowSize` (`boolean`)

EOActions XML Descriptions

APPLICATIONACTION

Description:

Actions sent to the application object.

XML Attributes:

`actionName` (`String`)

`actionPriority` (`int`)

`categoryPriority` (`int`)

`descriptionPath` (`String`)

`iconName` (`String`)

`iconURL` (`String`)

`menuAccelerator` (`String`—example “shift P”)

`shortDescription` (`String`)

`smallIconName` (n)

`smallIconURL` (`String`)

`standardAction` (`String`; if specified, all other arguments are ignored)

`toolTip` (`String`)

XMLClassValues

Superclass:

`Superclass`

Description:

Para

XML Tag:

XMLTagInfo

XMLClassValues

Superclass:

`Superclass`

Description:

Para

XML Tag:

XMLTagInfo

CONTROLLERHIERARCHYACTION

Description:

Actions dispatched throughout the controller hierarchy.

XML Attributes:

`actionName` (`String`)

`actionPriority` (`int`)

`categoryPriority` (`int`)

`descriptionPath` (`String`)

`iconName` (`String`)

`iconURL` (`String`)

`menuAccelerator` (`String`—example “shift P”)

`shortDescription` (`String`)

`smallIconName` (`String`)

`smallIconURL` (`String`)

`standardAction` (`String`; if specified, all other arguments are ignored)

`toolTip` (`String`)

HELPWINDOWACTION

Description:

Action in the Help menu to activate a window for a task.

XML Attributes:

`shortDescription` (`String`)

`menuAccelerator` (`String`—example “shift P”)

`multipleWindowsAvailable` (`boolean`)

`task` (String)

`toolTip` (String)

INSERTACTION

Description:

Action in the Document menu to insert an object of an entity.

XML Attributes:

`entity` (String)

`toolTip` (String)

OPENACTION

Description:

Action in the Document menu to open an object of an entity.

XML Attributes:

`entity` (String)

`toolTip` (String)

QUERYACTION

Description:

Action in the Document menu to start the search for an object of an entity.

XML Attributes:

`entity` (String)

`toolTip` (String)

SUPERCONTROLLERSACTION

Description:

Action to activate a window for a task.

XML Attributes:

`actionName` (String)

`actionPriority` (int)

`categoryPriority` (int)

`descriptionPath` (`String`)

`iconName` (`String`)

`iconURL` (`String`)

`menuAccelerator` (`String`—example “shift P”)

`shortDescription` (`String`)

`smallIconName` (`String`)

`smallIconURL` (`String`)

`standardAction` (`String`; if specified, all other arguments are ignored)

`toolTip` (`String`)

STANDARDACTION

XML Attributes:

`entity` (`String`)

TOOLWINDOWACTION

Description:

Action in the Tools menu to activate a windoe for a task.

XML Attributes:

`shortDescription` (`String`)

`menuAccelerator` (`String`—example “shift P”)

`multipleWindowsAvailable` (`boolean`)

`task` (`String`)

`toolTip` (`String`)

WINDOWACTION

Description:

Action to activate a window for a task.

XML Attributes:

`actionPriority` (`int`)

`categoryPriority` (`int`)

`categoryName` (String)

`descriptionPath` (String)

`iconName` (String)

`iconURL` (String)

`menuAccelerator` (String—example "shift P")

`multipleWindowsAvailable` (boolean)

`shortDescription` (String)

`smallIconName` (String)

`smallIconURL` (String)

`task` (String)

`toolTip` (String)

WINDOWOBSERVERACTION

Description:

Action sent to a window observer object (EOApplication.sharedApplication().windowObserver()).

XML Attributes:

`actionName` (String)

`actionPriority` (int)

`categoryPriority` (int)

`descriptionPath` (String)

`iconName` (String)

`iconURL` (String)

`menuAccelerator` (String—example "shift P")

`multipleWindowsAvailable` (boolean)

`shortDescription` (String)

`smallIconName` (String)

`smallIconURL` (String)

`standardAction` (String; if specified, all other arguments are ignored)

`toolTip` (String)

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

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 refer either to an attribute or to 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.

data modeling The process of building a data model to describe the mapping between a relational database schema and an object model.

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.

deep fetch An option available to fetch specifications that causes database fetches to occur against the root table and any leaf tables. Applicable to inheritance hierarchies.

derived attribute An attribute in a data model that does not directly correspond to a column in a database. Derived attributes are usually calculated from a SQL expression.

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 an HTML-based Web application from a model.

Direct to Web Assistant A tool 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 components.

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. It allows this data to persist in memory. See also key-value coding; property.

entity In Entity-Relationship modeling, a distinguishable object about which data is kept. An entity typically corresponds to a table in a relational database; an entity's attributes, in turn, correspond to a table's columns. An entity is used to map a relational database table to a Java class. See also attribute; table.

Entity-Relationship modeling A discipline for examining and representing the components and interrelationships in a database system. Also known as ER 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 Enterprise Objects to increase performance whereby destination objects of relationships are not fetched until they are explicitly accessed.

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

flattened attribute An attribute that is added from one entity to another by traversing a relationship.

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.

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

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.

inverse relationship A relationship that goes in the reverse direction of another relationship. Also known as a back relationship.

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 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.

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. In Enterprise Objects, a many-to-many relationship is composed of multiple relationships. 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.

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.

prefetching A feature in Enterprise Object that allows you to suppress fault creation for an entity's relationships. Instead of creating faults, the relationship data is fetched when the entity is first fetched. See also faulting.

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

prototype attribute An special type of attribute available in EOModeler to provide a template for creating attributes.

raw row fetching An possible option in a fetch specification that retrieves database rows without forming enterprise objects from those rows.

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.

reflexive relationship A relationship within the same entity; the relationship's source join attribute and destination join attribute are in the same entity.

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 foreign key; primary key; many-to-many relationship; to-many relationship; to-one relationship.

relationship key A key (an attribute) on which a relationship joins.

reusable component A component that can be nested within other components and acts like a dynamic element. Reusable components allow you to extend WebObjects 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.

snapshotting Part of the Enterprise Objects optimistic locking mechanism in which snapshots of database rows in memory are compared with the data in the database.

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 one corresponding destination record. For example, each employee has one job title.

transaction A set of actions that is treated as a single operation that either succeeds completely (COMMIT) or fails completely (ROLLBACK).

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.