Inside WebObjects

Java Client Desktop Applications

May 2002

 Apple Computer, Inc.
© 2000–2002 Apple Computer, Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Computer, Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Computer, Inc.
Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this book. Apple retains all intellectual property rights associated with the technology described in this book. This book is intended to assist application developers to develop applications only for Apple-labeled or Apple-licensed computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

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

Apple, the Apple logo, Mac, QuickTime, and WebObjects are trademarks of Apple Computer, Inc., registered in the United States and other countries.

Enterprise Objects and Enterprise Objects Framework are trademarks of NeXT Software, Inc., registered in the United States and other countries.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Figures, Listings, and Tables 11

Chapter 1 Introduction 19

Who Should Read This Book 20
Road Map 21
Related Documents 22
Java Client Features 22
  Better User Experience 22
  Object Distribution 23
  The Best of WebObjects 23
  Deployment Options 24
  Rapid Application Development 24
When to Choose Java Client 24
Java Client Development 27
Database Access 32
Java Client and Other Multi-Tier Systems 32

Chapter 2 Java Client Concepts 35

Enterprise Objects 35
  What Is an Enterprise Object? 36
  Enterprise Object Models 37
Java Client Architecture 37
Business Logic 39
Foundation Framework 40
Access Layer 41
  Essential EOAccess Classes 41
    EOAdaptor 42
    EODatabaseContext 42
    EOModel 42
    EOUtilities 42

© Apple Computer, Inc. May 2002

Control Layer 43
Essential EOControl Classes 43
EOEnterpriseObject 43
EOEditingContext 44
EOFetchSpecification 46
EOGlobalID 46
EOObjectStoreCoordinator 46
Distribution Layer 46
Essential EODistribution Classes 47
Client Interface Layer 48
Display Groups 48
Associations 48
Application Layer 50
Generation Layer 50
Model-View-Controller Paradigm 52
Deploying and Using Java Client Applications 53
Server Requirements 54

Chapter 3 Basic Tutorial 55

Create the Database 56
Create an EOModel 57
Behind the Steps 59
Build the Model 61
Behind the Steps 63
Completing the Model 64
Behind the Steps 66
Generate SQL 67
Behind the Steps 68
Create the Project 69
Behind the Steps 71
More About The Java Client Class Loader 72
The Default Project 73
Groups 73
Targets 74
Client Files (Web Server Target) 75
Server Files (Application Server Target) 75

© Apple Computer, Inc. May 2002

C O N T E N T S

Add a Launch Argument 77
  More About Session Time Outs 79
Build the Executable 79
Run the Client Application 80
  Prepare to Run the Project 80
  Client Launch Script 81
    Behind the Steps 82
  Java 82
  JDK appletviewer 82
  MRJ Application 83
Application Startup 83
Using the Application 84
  Behind the Steps 88
Customizing the Application 88
  Behind the Steps 94
Inside Assistant 95
  Entities 95
    Main Entities 95
    Enumeration Entities 95
    “Other” Entities 96
  Properties 96
    Task 96
    Question 98
    Property Keys 98
  Widgets 99
  Windows 99
  Miscellaneous 99
  XML 99
Add a Relationship 99
  Add an Entity 99
  Make the Relationship 101
  The Enhanced Application 104
Where to Go From Here 106

© Apple Computer, Inc. May 2002

Chapter 4 Distribution Layer 107

Business Logic Partitioning 107
  Design Recommendations 108
  Performance 109
Remote Method Invocations 110
  On Business Logic 110
  On Application Logic 111
  Distributed Object Store 112
Custom Code in Business Logic 113
Distribution Layer Objects 114
Data Synchronization 116
Distribution Channels 117
Delegates 117

Chapter 5 Advanced Tutorial 119

Customization Techniques 119
Enhance the EOModel 122
  Add an Entity 123
  Make a Relationship 125
Add Custom Business Logic 128
  Prepare the Project for Custom Logic 129
  Generate Source Files 130
    Behind the Steps 133
  Prepare Application for Business Logic 134
  Add Custom Code 138
    Validation 140
    Initial Values 142
Controller Hierarchy 144
  Controllers 144
  Creating the Controller Hierarchy 145
Using Rules in the Rule System 146
Additional Actions 148
  Write the Action 148
  Use Assistant 153
  Extend a Controller Class 158
  Additional Exercise 161
Debugging 165

© Apple Computer, Inc. May 2002

Chapter 6 Nondirect Java Client Development 167

Building Custom Interfaces 167
Laying Out the User Interface 171
Prepare the Nib File 172
Integrate the Model 174
Add Formatters 177
Adding Action Methods 179
Create a Master-Detail Interface 181
Build and Run 185
Programmatic Access to Interface Components 185
Cocoa to Swing Translation 187

Chapter 7 Inside the Rule System 189

How It Works 189
Rule System Priorities 191
D2WComponents 191
Rule System Requests 191
Internal Rule System Requests 192
Generating the Student Form Window 193
EOSwitchComponent 196

Chapter 8 Task: Restricting Access to an Application 199

The Documents Menu 199
The Default Query Window 200
Restricting Tasks Within the Application 201

Chapter 9 Task: Using the Controller Factory Programmatically 205

Selecting Objects in an Entity 205
Triggering a Task 206
Inserting Objects 207

© Apple Computer, Inc. May 2002

Chapter 10 Task: Adding Custom Actions to Controllers 209

Subclassing Controller Classes 209
Writing Custom Controller Classes 213

Chapter 11 Task: Adding Custom Menu Items 215

About Actions 215
New D2WComponent 216
Application-Wide Actions 217
Menu-Specific Actions 218
Controller-Specific Actions 219

Chapter 12 Task: Customizing With Common Rules 221

Confirmation Dialog 221
Window Size 222
Widget Alignment 223
Custom Controllers 223
Custom Class for Widgets 224
Custom Attributes for Controllers 225

Chapter 13 Task: Freezing XML User Interfaces 227

Freeze XML User Interfaces 227
  Customize the XML 234
Adding Actions to Frozen XML 236
  Edit XML by Hand 237
  Using a Custom Controller Class in Frozen XML 237

Chapter 14 Task: Mixing Static and Dynamic User Interfaces 241

Preparing the Nib for Freezing 241
Integrating the Nib File 243

© Apple Computer, Inc. May 2002

C O N T E N T S

Chapter 15 Task: Using Custom Views in Interface Files 245

Custom Views 245
EOImageView 253

Chapter 16 Task: Localizing Dynamic Components 255

Localizing Property Labels 255
Localizing the Standard Strings and Frozen XML Components 262

Chapter 17 Task: Building Custom List Controllers 265

Chapter 18 Task: Using and Extending Image Views 267

Adding Outlets 267
Adding the Widget 268
Connecting the Outlet 270
Loading the Image 271

Chapter 19 Task: Using Pop-up Menus In Nib Files 275

Chapter 20 Task: Building a Login Window 285

Building the User Interface 285
Adding Logic to Authenticate Users 290
Restricting Access 294

Appendix A XML Description of Classes and Actions 297

XML Value Types 297
Classes With XML Tags and XML Attributes 299
EOActions XML Descriptions 312

© Apple Computer, Inc. May 2002

Glossary 315

Index 321

© Apple Computer, Inc. May 2002

Figures, Listings, and Tables

Chapter 1 Introduction 19

Figure 1-1 A custom Java Client interface 29
Figure 1-2 A typical Direct to Java Client application 29
Figure 1-3 Dynamically generated user interface 30
Table 1-1 Comparison of static and dynamic user interfaces 31

Chapter 2 Java Client Concepts 35

Figure 2-1 Java Client architecture 38
Figure 2-2 Editing contexts and object stores 45
Figure 2-3 The complete stack of WebObjects layers in Direct to Java Client 51

Chapter 3 Basic Tutorial 55

Figure 3-1 Part of the completed application in this chapter 56
Figure 3-2 Configuring a new database 57
Figure 3-3 JDBC connection information 58
Figure 3-4 Deselect all options for this model 59
Figure 3-5 Entity Inspector 62
Figure 3-6 Name Attribute Inspector 63
Figure 3-7 The primary key attribute 65
Figure 3-8 The finished model 65
Figure 3-9 Generate SQL 67
Figure 3-10 Choose EOModel 70
Figure 3-11 Configure the class loader 71
Figure 3-12 Target pop-up menu 75
Figure 3-13 The default groups and files 77
Figure 3-14 Add session timeout launch argument 78
Figure 3-15 Default enumeration window 85
Figure 3-16 Revised model 86

© Apple Computer, Inc. May 2002

Figure 3-17 Schema Synchronization window 87
Figure 3-18 Revised enumeration window 88
Figure 3-19 Change entity type 89
Figure 3-20 Query window with data 90
Figure 3-21 Query window searching for names containing "e" 90
Figure 3-22 Properties tab in Assistant 91
Figure 3-23 Query on GPA 92
Figure 3-24 Left-hand side of rules 93
Figure 3-25 Right-hand side of rules 93
Figure 3-26 The application with simple customizations 94
Figure 3-27 Form window 97
Figure 3-28 Query window and list task 98
Figure 3-29 Activity entity 100
Figure 3-30 Relate Student and Activity 101
Figure 3-31 Relationship Inspector for Student's activities relationship 102
Figure 3-32 Relationship Inspector for Activity's student relationship 103
Figure 3-33 Make Student to Activity relationship a client-side class property 104
Figure 3-34 Do not make Activity to Student relationship a client-side class property 104
Figure 3-35 Add activities to new Student record 105

Chapter 4 Distribution Layer 107

Figure 4-1 Objects in the distribution layer 115

Chapter 5 Advanced Tutorial 119

Figure 5-1 The updated Student entity 123
Figure 5-2 Interview entity 124
Figure 5-3 Generate SQL for the Interview entity 125
Figure 5-4 The interviews relationship in the Student entity 126
Figure 5-5 The student relationship in the Interview entity 127
Figure 5-6 Student's relationships 127
Figure 5-7 Interview's relationship 128
Figure 5-8 Directory structure for custom business logic 130

© Apple Computer, Inc. May 2002

Figure 5-9 Save Client Java files in BusinessLogic/Client 131
Figure 5-10 Import BusinessLogic directory 132
Figure 5-11 BusinessLogic group with imported files and associated targets 133
Figure 5-12 Add a property key for the form task 134
Figure 5-13 Additional property key for list task 135
Figure 5-14 Change the widget type to make the association. 136
Figure 5-15 Change formatter for property in list view 137
Figure 5-16 The rating field in action 140
Figure 5-17 Initial values 143
Figure 5-18 Validation exception message 143
Figure 5-19 New key of type Student in the Report component 149
Figure 5-20 New key of type Interview in the Report component 150
Figure 5-21 New key of type Activity in the Report component 151
Figure 5-22 Dynamic elements for Student's attributes 151
Figure 5-23 WORepetition for Student's interviews 152
Figure 5-24 WORepetition for Student's activities 152
Figure 5-25 Add property key for new action 154
Figure 5-26 Change the widget type of the new property key 155
Figure 5-27 The new property key as an EOActionController 156
Figure 5-28 Add launch argument for SMTP host 158
Figure 5-29 Image form window with new buttons 161
Figure 5-30 Choose email recipients 164
Listing 5-1 CustomFormController code 159
Table 5-1 Consequences of each customization technique 122
Table 5-2 A subset of the controllers available in Direct to Java Client 145

Chapter 6 Nondirect Java Client Development 167

Figure 6-1 Name the interface controller 168
Figure 6-2 Choose a template for the interface controller 169
Figure 6-3 Interface Builder palettes 170
Figure 6-4 Enterprise Objects palette 170
Figure 6-5 The Interface Builder environment 172
Figure 6-6 Classes pane in the nib file window 173
Figure 6-7 Assign the custom subclass to File's Owner 174
Figure 6-8 The Student entity dragged into Interface Builder 175

© Apple Computer, Inc. May 2002

Figure 6-9 Display group and editing context 175
Figure 6-10 Display group options in Interface Builder 176
Figure 6-11 Choose a formatter for the Gpa column 177
Figure 6-12 Choose a formatter for the FirstContact column 178
Figure 6-13 Testing the application 179
Figure 6-14 Connect the Add button to the insert method of the Student EODisplayGroup 180
Figure 6-15 Select the insert method 181
Figure 6-16 The activities relationship in the Student entity 182
Figure 6-17 A master-detail interface 183
Figure 6-18 Complete widget set for the master-detail interface 184
Figure 6-19 Connect widgets with associations 185
Figure 6-20 Add an outlet 186
Figure 6-21 Connect the new outlet 187

Chapter 8 Task: Restricting Access to an Application 199

Figure 8-1 Default actions in a form window 201
Figure 8-2 Disabled actions in a form window 203

Chapter 9 Task: Using the Controller Factory Programmatically 205

Figure 9-1 Select dialog 206
Figure 9-2 Form window from controller factory 208

Chapter 10 Task: Adding Custom Actions to Controllers 209

Figure 10-1 Image form window with new actions 212
Listing 10-1 Subclassing EOFormController 210
Listing 10-2 A custom controller class 213

Chapter 11 Task: Adding Custom Menu Items 215

Listing 11-1 Changing the superclass of UserActions 217

© Apple Computer, Inc. May 2002

Chapter 12 Task: Customizing With Common Rules 221

Figure 12-1 Confirm dialog on unqualified queries 221

Chapter 13 Task: Freezing XML User Interfaces 227

Figure 13-1 Select Component as the file type 228
Figure 13-2 Name new component "StudentFormWindow" 229
Figure 13-3 XML description of Student entity, form window 231
Figure 13-4 Make a new rule file for custom rules 232
Figure 13-5 Add a rule to use frozen XML 233
Figure 13-6 Student form window with BOXCONTROLLER tag 236
Figure 13-7 Action in custom controller class 239
Listing 13-1 Change the superclass of StudentFormWindow to D2WComponent 230
Listing 13-2 StudentFormWindow.html (frozen XML) 235

Chapter 14 Task: Mixing Static and Dynamic User Interfaces 241

Figure 14-1 Classes pane in the nib file window 242
Figure 14-2 Assign the custom subclass to File's Owner 243

Chapter 15 Task: Using Custom Views in Interface Files 245

Figure 15-1 Custom view object in window 246
Figure 15-2 Find NSView in class hierarchy 246
Figure 15-3 Name the custom view class 247
Figure 15-4 Associate custom view with NSView subclass 248
Figure 15-5 Custom view as NSView subclass 249
Figure 15-6 File's Owner's class 249
Figure 15-7 Add outlet to interface file 250
Figure 15-8 Connect new outlet to custom view 251
Figure 15-9 File's Owner's attributes 252

© Apple Computer, Inc. May 2002

Chapter 16 Task: Localizing Dynamic Components 255

Figure 16-1 Right-hand side class of type Assignment 255
Figure 16-2 Right-hand class of type Custom 256
Figure 16-3 Add localized variant of Localizable.strings file 259
Figure 16-4 Add localized variant for German 260
Figure 16-5 Localized resources in project 260
Listing 16-1 LocalizedStringLookup class 256
Listing 16-2 German-localized variants of strings file 261

Chapter 18 Task: Using and Extending Image Views 267

Figure 18-1 Add a new outlet 268
Figure 18-2 Cocoa-Other palette 269
Figure 18-3 Place widget with guides 269
Figure 18-4 File’s Owner icon with exclamation point 270
Figure 18-5 Connect outlet to widget 271
Figure 18-6 Image in image view 273
Listing 18-1 Overriding controllerDidLoadArchive 273

Chapter 19 Task: Using Pop-up Menus In Nib Files 275

Figure 19-1 Illustrator entity in nib file 275
Figure 19-2 Cocoa-Other palette 276
Figure 19-3 Connect widget to display group 277
Figure 19-4 Bind the title aspect to the appropriate attribute 278
Figure 19-5 EODisplayGroup object in nib file 279
Figure 19-6 Bind File’s Owner’s controllerDisplayGroup outlet 279
Figure 19-7 Bind the outlet 280
Figure 19-8 Add a key to display group 281
Figure 19-9 Bind selectedIndex attribute of association to display group key 282
Figure 19-10 A pop-up menu in action 283

© Apple Computer, Inc. May 2002

Chapter 20 Task: Building a Login Window 285

Figure 20-1 Login window user interface 286
Figure 20-2 Add outlets named username and password 286
Figure 20-3 Add actions 287
Figure 20-4 File's Owner with new connections 288
Figure 20-5 Select the WOJavaClientApplet dynamic element 289
Figure 20-6 Add value for interfaceControllerClassName binding 290
Figure 20-7 Login failed 293
Listing 20-1 Client-side login method 291
Listing 20-2 login method 292
Listing 20-3 Authentication in Session.java 292
Listing 20-4 Load a nib file programmatically 295

© Apple Computer, Inc. May 2002

© Apple Computer, Inc. May 2002

CHAPTER 1

Introduction

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 run 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 5.1. Future versions of WebObjects may include API 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 features to applications like access controls, custom menu items, and sophisticated user interfaces.

© Apple Computer, Inc. May 2002

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. Since 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.

© Apple Computer, Inc. May 2002

If you're new to WebObjects development, you may find the book *Inside WebObjects: Discovering WebObjects for HTML* useful when learning Java Client as it provides an introduction to the WebObjects tools and to common WebObjects programming techniques and concepts. Furthermore, the book *Inside WebObjects: Developing WebObjects Applications With Direct to Web* 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.

Road Map

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 "Basic Tutorial" (page 55) 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 "Advanced Tutorial" (page 119), 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 of the book 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 approaches the topic of Java Client applications in a way different from that of 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 27) for more information on this topic.

© Apple Computer, Inc. May 2002

Related Documents

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/techpubs/webobjects
- The WebObjects CD-ROM, which contains the WebObjects API reference, various documents in HTML and PDF, examples, what's new, and legacy documentation

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 allows Java Client applications to take advantage of the rich user interface elements the Swing toolkit offers. This is perhaps the primary reason why you'd choose to build a WebObjects application using Java Client: the need for a rich, more interactive user interface.

Rich user interfaces allow you to 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.

© Apple Computer, Inc. May 2002

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. It is up to the developer to control how this distribution occurs.

In all multi-tier 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 can quite easily be 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.

© Apple Computer, Inc. May 2002

Deployment Options

The client-side application of WebObjects Java Client applications runs on any JDK 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 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 staring 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 can take advantage of 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

© Apple Computer, Inc. May 2002

ideal, however, for use in uncontrolled Internet environments or for high-traffic websites. Typically, Java Client applications, when deployed as desktop 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. Deploying as applets alleviates many of the issues encountered when running Java Client applications as desktop applications since users don't need to download or install the client application. However, applets introduce other usability and deployment issues. See "Deploying and Using Java Client Applications" (page 53) for a comparison of the two deployment methods.

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 72) for more information.

Likewise, new features of the JDK such as Web Start ease application deployment and usability by providing caching and other mechanisms to ease client-side class management.

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.

© Apple Computer, Inc. May 2002

- Portability. Java Client applications are 100% Pure Java applications, requiring a JRE (Java Runtime Environment) 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 instance, 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 simple user interface actions such as this. 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 some customers will be put off by the proposition of installing the JRE.
- 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 snappier than their Web counterparts.

© Apple Computer, Inc. May 2002

- 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. In code-specific terms, Direct to Java Client applications are instances of `com.webobjects.eoapplication.EODynamicApplication` whereas nondirect Java Client applications are instances of `com.webobjects.eoapplication.EOApplication`.

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

© Apple Computer, Inc. May 2002

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

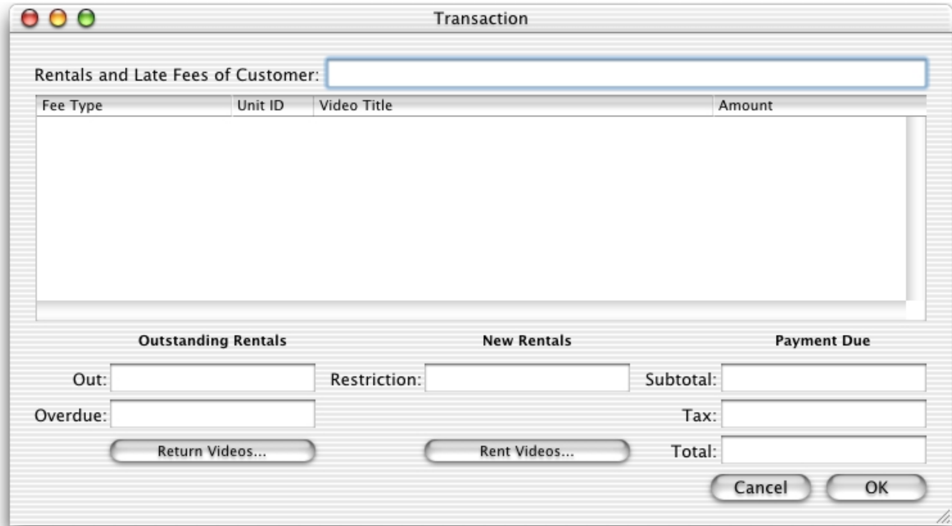
This book covers all of these customization methods.

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 1-1.

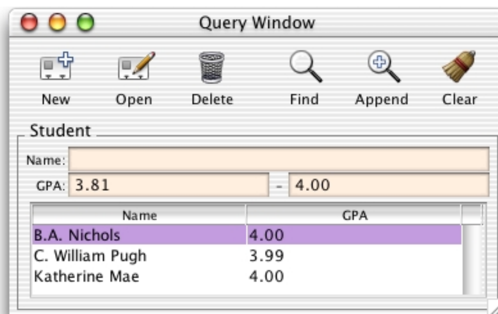
© Apple Computer, Inc. May 2002

Figure 1-1 A custom Java Client interface



The dynamic user-interface generation provided in Direct to Java Client applications yields interfaces that resemble Figure 1-2. 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.

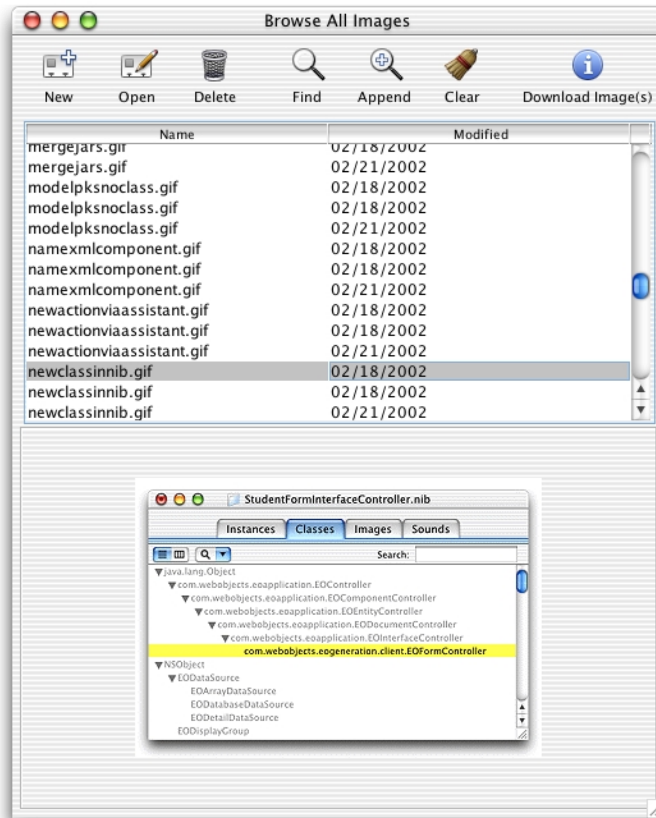
Figure 1-2 A typical Direct to Java Client application



© Apple Computer, Inc. May 2002

Figure 1-3 shows dynamically-generated user interfaces that make use of custom controller classes, custom rules, and programmatic invocations of the controller factory.

Figure 1-3 Dynamically generated user interface



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

© Apple Computer, Inc. May 2002

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 1-1.

Table 1-1 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.
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

© Apple Computer, Inc. May 2002

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

See "Related Documents" (page 22) to learn how to access the WebObjects API reference and other documents on the Enterprise Object technology.

Java Client and Other Multi-Tier Systems

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

© Apple Computer, Inc. May 2002

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 has the last word on what data is committed).

© Apple Computer, Inc. May 2002

© Apple Computer, Inc. May 2002

C H A P T E R 2

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. Chapter 3, "Basic Tutorial," 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 several 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.

© Apple Computer, Inc. May 2002

- **com.webobjects.eointerface** mediates between the control layer and an application's user interface; maps data to user interface elements.
- **com.webobjects.eodistribution, com.webobjects.eodistribution.client** distributes enterprise objects across the network to the client; provides much of the functionality of the EOAccess layer on the client.
- **com.webobjects.eoapplication** is a general user-interface utility layer specific to both types of Java Client applications.
- **com.webobjects.eogeneration, com.webobjects.eogeneration.client** dynamically generates 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.

© Apple Computer, Inc. May 2002

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.)

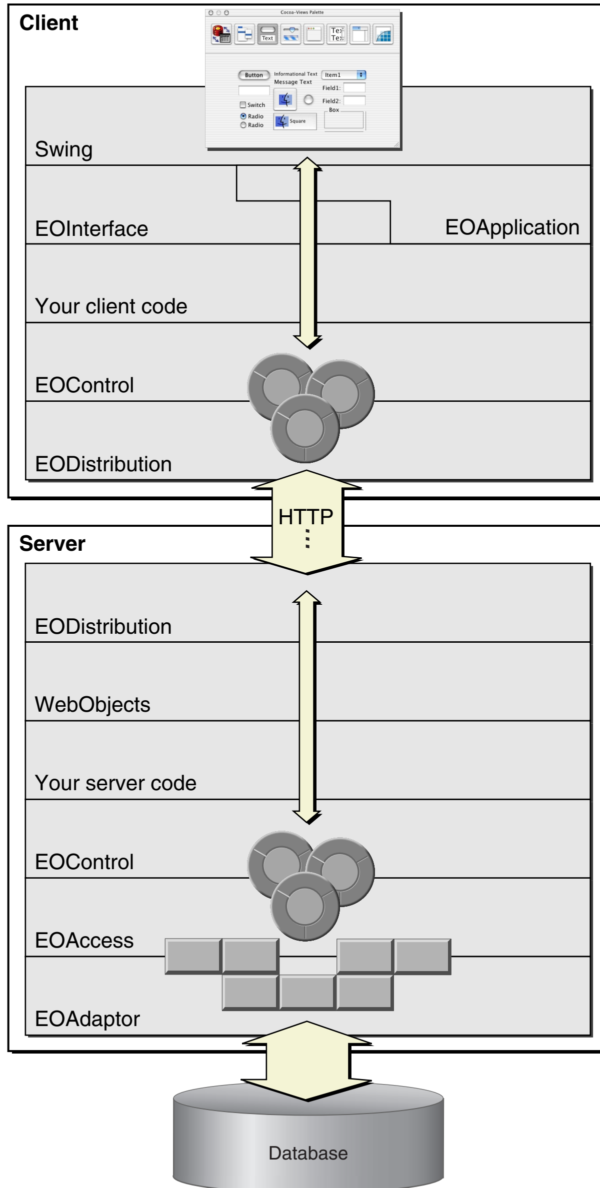
Java Client Architecture

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 (page 38) illustrates this architecture.

© Apple Computer, Inc. May 2002

Figure 2-1 Java Client architecture



© Apple Computer, Inc. May 2002

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 Multi-Tier Systems" (page 32).

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.

© Apple Computer, Inc. May 2002

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 107).

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.

© Apple Computer, Inc. May 2002

- **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; 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 programmatically interact with data stores in an abstract manner. It is designed to be data store-agnostic, so many of its objects are reusable. 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.

© Apple Computer, Inc. May 2002

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

This 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

This class 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.

© Apple Computer, Inc. May 2002

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, 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").
  ```


© Apple Computer, Inc. May 2002

- **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.

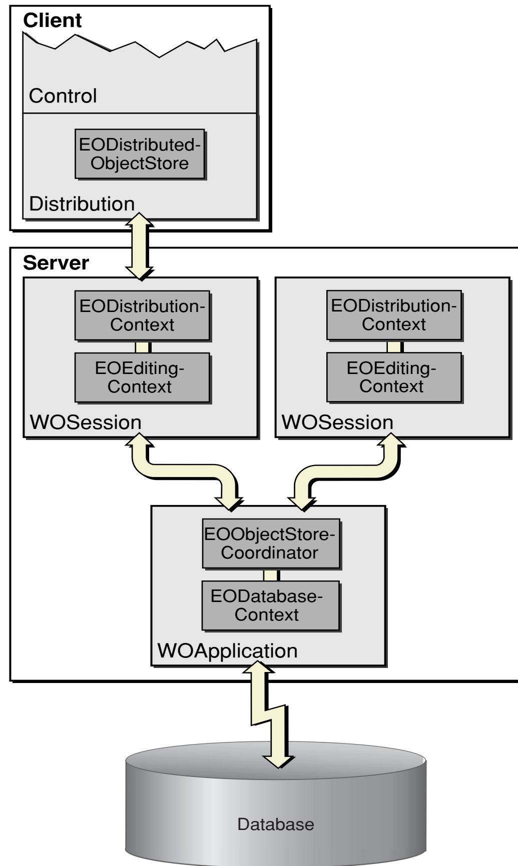
An EOEditingContext is always associated with an instance of a parent object store. In Java Client applications, the client and server have separate editing contexts. The client-side editing context is associated with a client-side object store, `com.webobjects.eodistribution.client.EODistributedObjectStore`; the server-side editing context is associated with a server-side object store, `com.webobjects.eoaccess.EODatabaseContext`, as illustrated in Figure 2-2 (page 45).

You can think of an EOEditingContext object as a glorified 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

© Apple Computer, Inc. May 2002

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.

Figure 2-2 Editing contexts and object stores



© Apple Computer, Inc. May 2002

EOFetchSpecification

Because database fetches are 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 a 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.

© Apple Computer, Inc. May 2002

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 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.
- WOJavaClientApplet. 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 Chapter 4, "Distribution Layer" (page 107), for more information on the distribution layer and to better understand how these objects work together.

© Apple Computer, Inc. May 2002

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 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.

An EOAssociation has aspects that define the different parameters of the display object it controls. 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.

© Apple Computer, Inc. May 2002

- `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 TableColumnAssociations 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.

© Apple Computer, Inc. May 2002

Application Layer

The EOApplication layer, defined in `com.webobjects.eoapplication`, isolates the developer from the idiosyncrasies of each execution environment. 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.

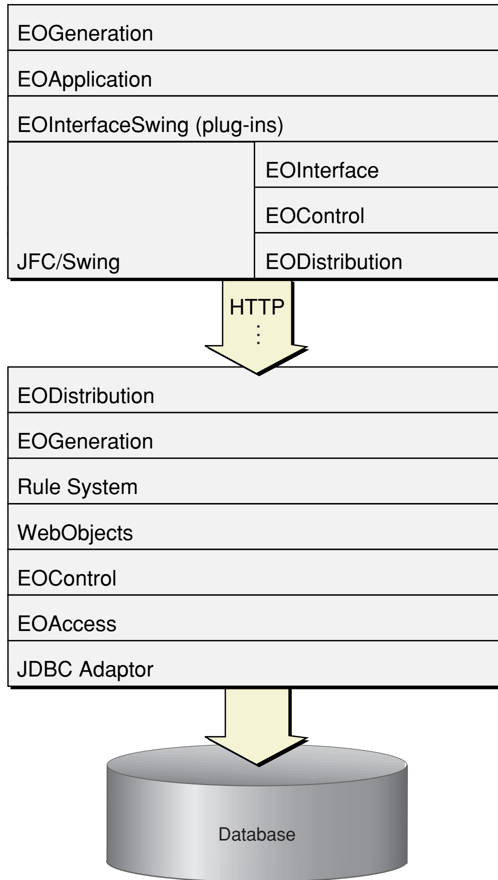
Generation Layer

The EOGeneration layer, defined in `com.webobjects.eogeneration` and `com.webobjects.eogeneration.client`, dynamically generates user interfaces in Java Client applications which 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, along with the WebObjects rule system, are the elements that make a Direct to Java Client application different from a nondirect Java Client application. They are illustrated in Figure 2-3 (page 51).

© Apple Computer, Inc. May 2002

Figure 2-3 The complete stack of WebObjects layers in Direct to Java Client



© Apple Computer, Inc. May 2002

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.

Note: "Model" does not have the same meaning in WebObjects as it does in the MVC paradigm. In WebObjects, a model (short for "EOModel") establishes and maintains correspondance between an enterprise object class and data stored in a relational database. In MVC, model objects represent the special knowledge of the application.

View objects represent things visible on 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

© Apple Computer, Inc. May 2002

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.

Deploying and Using Java Client Applications

HTML-based WebObjects applications require only a Web browser on the client. The client requirements for Java Client desktop applications, however, are considerably more demanding.

Java Client applications can be deployed as either applets running within a browser or as real desktop applications. Either deployment option is feasible, but you should carefully evaluate both options after understanding their respective strengths and weaknesses.

Installation
: Applets require no installation of the Java Client application on the user's part, since the Web browser handles the downloading of classes. Applications, however, need to be installed on the client.

Upgrades
: Using applets, 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
: Both applications and applets 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 plugin 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 either applets or as full desktop applications.

© Apple Computer, Inc. May 2002

User experience

Running Java Client applications as desktop applications 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. Java Client applications running as desktop 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, 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 should be an insignificant factor in choosing a deployment method.

Security

From the user's perspective, running as applets is inherently more secure, since applets prevent the JVM from accessing the file system or other parts of the user's system. Developers usually prefer applications over applets because they don't have to worry about the security restrictions inherent to Web browsers.

Note: A new technology in the JDK, Web Start, alleviates the issues with application installation and upgrades. It allows users to access applications from hyperlinks in Web browsers. The hyperlink invokes Web Start on the client, which takes care of downloading all the classes needed for the client-side application.

Server Requirements

Serving Java Client applications requires just a WebObjects application server. You use the standard WebObjects deployment tools and techniques. See the book *Inside WebObjects: Deploying WebObjects Applications* for complete documentation.

© Apple Computer, Inc. May 2002

Basic Tutorial

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

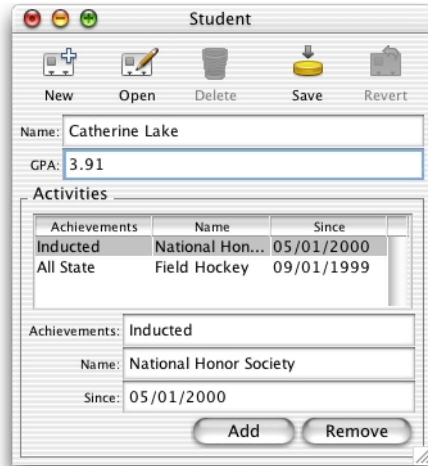
- create a simple database using OpenBase Manager
- create tables in that database using EOModeler
- build a Direct to Java Client application using Project Builder
- perform simple customizations of the application using the Direct to Java Client Assistant

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 (page 56) shows a sample student record from this application.

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

© Apple Computer, Inc. May 2002

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



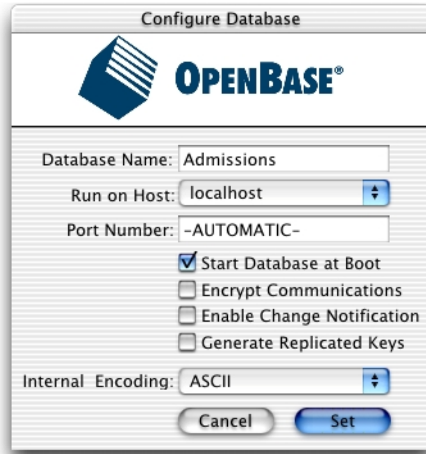
Create the Database

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 ASCII for the Internal Encoding pop-up menu. The Configure Database dialog should appear as shown in Figure 3-2 (page 57).

© Apple Computer, Inc. May 2002

Figure 3-2 Configuring a new database



4. Click Set.
5. You may have to 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) before moving to the next step.
6. Quit OpenBase Manager.

Create an EOModel

EOModeler is a powerful application that provides tools to build and manage your business logic. 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

© Apple Computer, Inc. May 2002

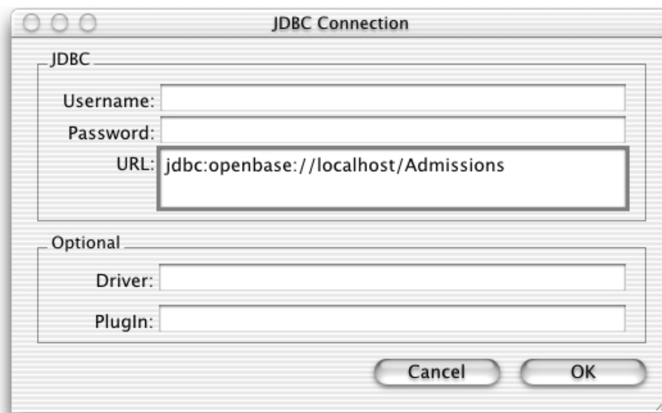
- generate SQL
- generate client and server Java files based on EOModels
- build fetch specifications

A good model is important because Direct to Java Client's generation layer analyzes EOModels to generate user interfaces. 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 the following in the URL field: `jdbc:openbase://localhost/Admissions`, as shown in Figure 3-3. Click OK.

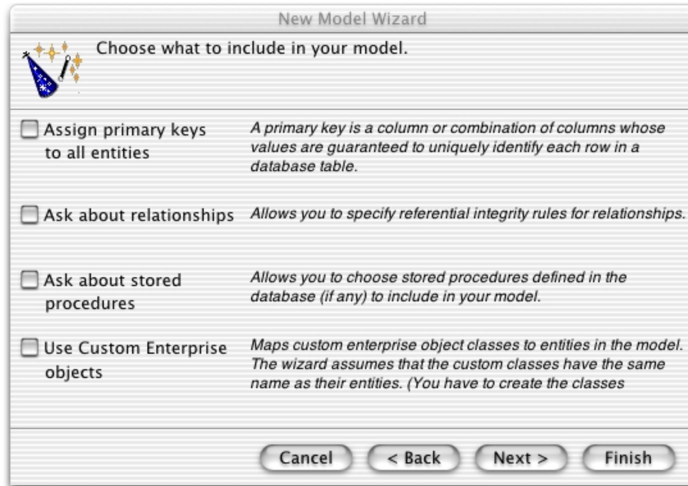
Figure 3-3 JDBC connection information



5. Since the database is empty, deselect the four options in the next window and click Next. See Figure 3-4 (page 59).

© Apple Computer, Inc. May 2002

Figure 3-4 Deselect all options for this model



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, and MySQL are qualified for WebObjects 5.1. See the document *Post-Installation Guide* for more exact specifications. Third parties have developed JDBC adaptors for other JDBC-compliant databases. See the Apple Support Knowledge Base for information on creating custom JDBC adaptors.

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 5.1 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.

© Apple Computer, Inc. May 2002

- "Assign primary keys to all entities"—When reading and writing to databases, the EOAccess 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).

© Apple Computer, Inc. May 2002

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.

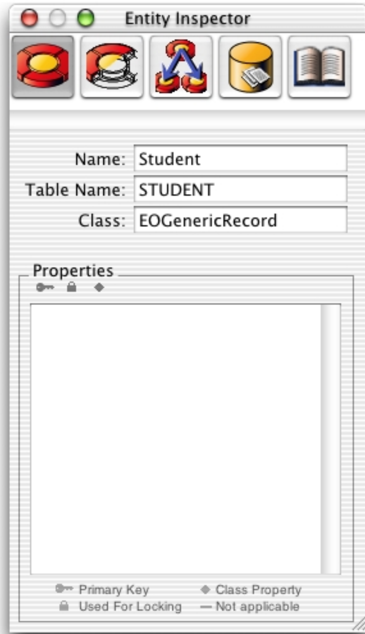
Note: The model you'll create will initially be suboptimal so that the tutorial can demonstrate some features of Java Client you wouldn't otherwise see with a perfect model.

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.
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 3-5 (page 62).

© Apple Computer, Inc. May 2002

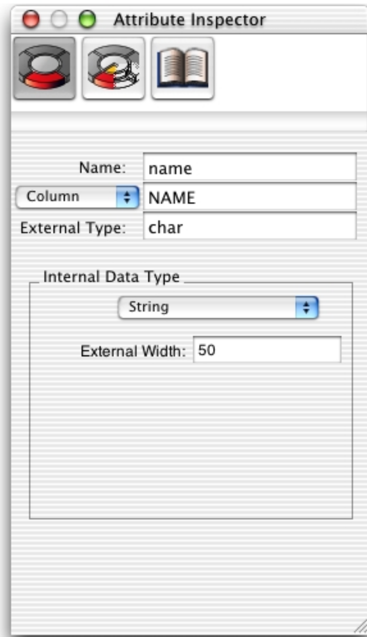
Figure 3-5 Entity Inspector



4. Add a new attribute 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 3-6 (page 63).

© Apple Computer, Inc. May 2002

Figure 3-6 Name Attribute Inspector



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."

© Apple Computer, Inc. May 2002

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.

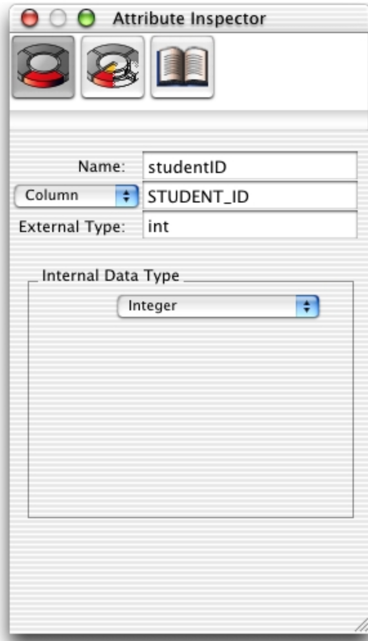
Completing the Model

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 an internal data type of `Integer`. This attribute will be the entity's primary key. See Figure 3-7 (page 65).

© Apple Computer, Inc. May 2002

Figure 3-7 The primary key attribute



2. In table mode (Tools > Table Mode), you'll notice three icon fields next to each attribute. The key icon denotes a primary key, the diamond denotes a server-side class property, and the lock denotes the attribute is used for locking. Make the `studentID` attribute the primary key by clicking in the key field next to it.

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

Figure 3-8 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 ▼	

© Apple Computer, Inc. May 2002

4. To select which attributes are sent to the client, you need to add a view column in EOModeler. Click the Add Column pop-up menu and select Client-Side Class Property. This adds a column with two opposing arrows to the icon fields. Make sure that only the `gpa` and `name` attributes are selected as client-side class properties, as shown in Figure 3-8.
5. Save the model as "Admissions.eomodeld."

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, and 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.

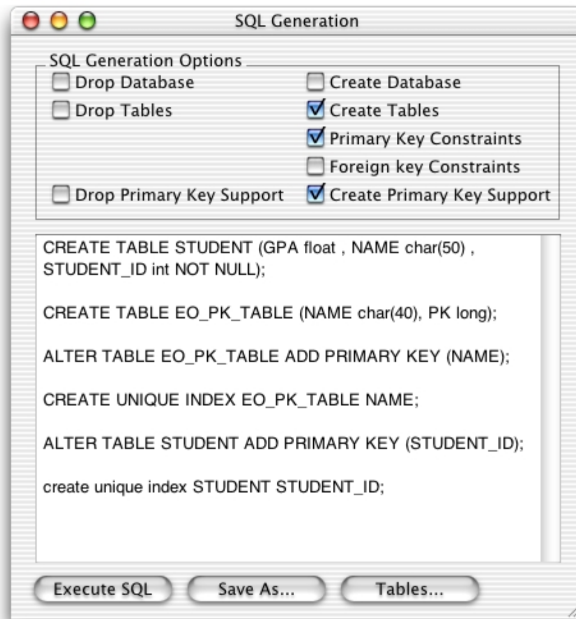
© Apple Computer, Inc. May 2002

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 entities list.
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 3-9 (page 67).
4. Click Execute SQL.

Figure 3-9 Generate SQL



© Apple Computer, Inc. May 2002

5. To verify the table was written to the database, in OpenBase Manager, select 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`.
- **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.

© Apple Computer, Inc. May 2002

Create the Project

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

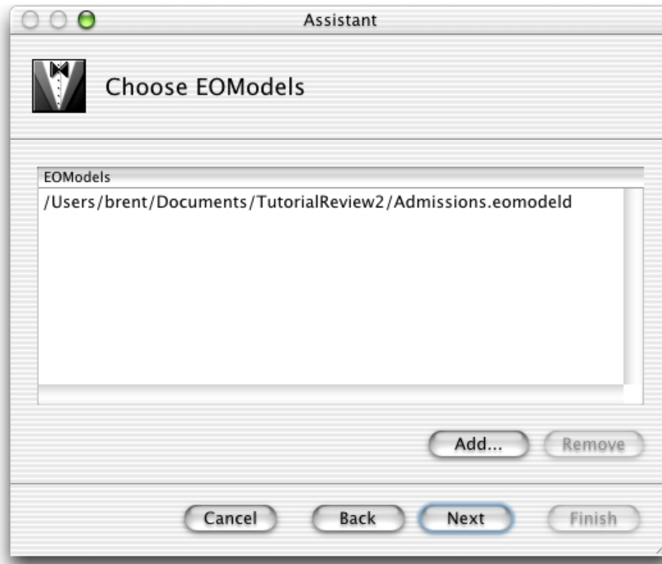
- creating working projects from project templates
- organizing project files and resources
- source-code editing
- compiling and debugging projects
- running projects
- communicating with other WebObjects development tools

Project Builder 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 under the WebObjects group as the new project type.
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. See Figure 3-10 (page 70).

© Apple Computer, Inc. May 2002

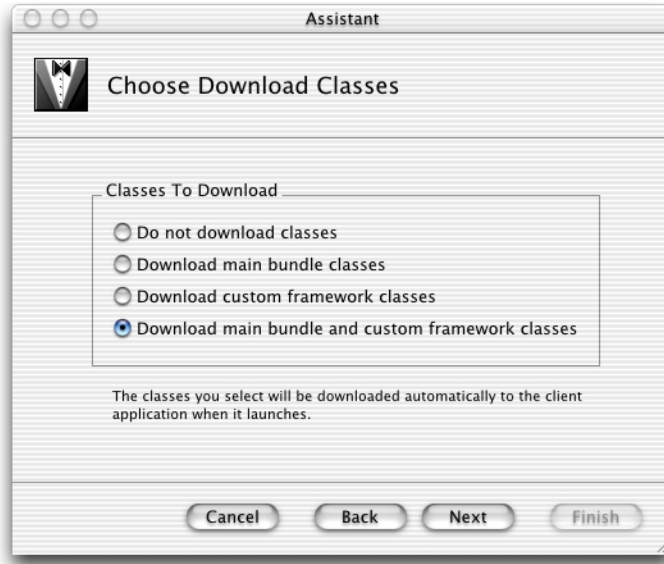
Figure 3-10 Choose EOModel



9. In the Choose Download Classes pane, select the option “Download main bundle and custom framework classes” and click Next. See Figure 3-11 (page 71).

© Apple Computer, Inc. May 2002

Figure 3-11 Configure the class loader



10. 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 77) 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 84). 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.

© Apple Computer, Inc. May 2002

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. There are four 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 WOJavaClientApplet component of your 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`

© Apple Computer, Inc. May 2002

- `customFrameworksClientClasses`
- `customBundlesClientClasses`

This feature is useful for deployment (since installing an update of the client desktop application is only necessary when you switch the version of WebObjects you use, but 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 only useful for organizational purposes: they have no effect on how their content or the application behaves.

© Apple Computer, Inc. May 2002

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, JPEG) 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.

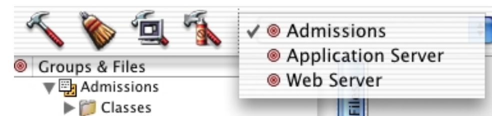
The Web Server and Application Server targets are build targets and the Admissions target (or the target named after your application) is the root or aggregate target.

© Apple Computer, Inc. May 2002

- **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 3-12 shows.

Figure 3-12 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 is the Main component (`Main.wo`) in the Main subgroup located in the Web Components group.

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

```
<HTML>
<HEAD>
  <TITLE>Main</TITLE>
</HEAD>
```


© Apple Computer, Inc. May 2002

```
<BODY>
 <CENTER>Please wait for the application to open other windows (please note:
it is better to start the application from the command line, as
a java application).<WEBOBJECT NAME=Applet></WEBOBJECT></CENTER>
</BODY>
</HTML>
```

The `Main.wod` file contains this code:

```
Applet: WOJavaClientApplet {
    applicationClassName =
          "com.webobjects.eogeneration.client.EODynamicApplication";
    height = 0;
    width = 0;
    useJavaPlugin = true;
    downloadClientClasses = "customBundlesClientClasses";
}
```

The `<WEBOBJECT NAME=Applet>` tag in `Main.html` is bound to the definition of `Applet` in `Main.wod`, which specifies that `Applet` represents a `WOJavaClientApplet` component. In `Main.wod`, some bindings for `WOJavaClientApplet` are specified.

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.eoapplication.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). See "Distribution Layer Objects" (page 114) for more information on the bindings for WOJavaClientApplet.

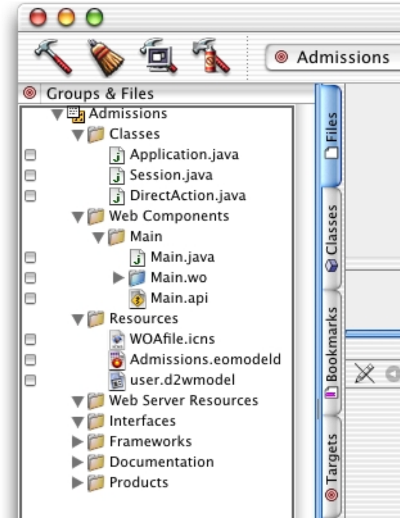
Other server files include

- 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 3-13 (page 77) shows the default groups and files.

© Apple Computer, Inc. May 2002

Figure 3-13 The default groups and files



The next section continues building the tutorial project.

Add a Launch Argument

Java Client applications have usability patterns different from those of HTML applications—their usage patterns resemble those of desktop 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.

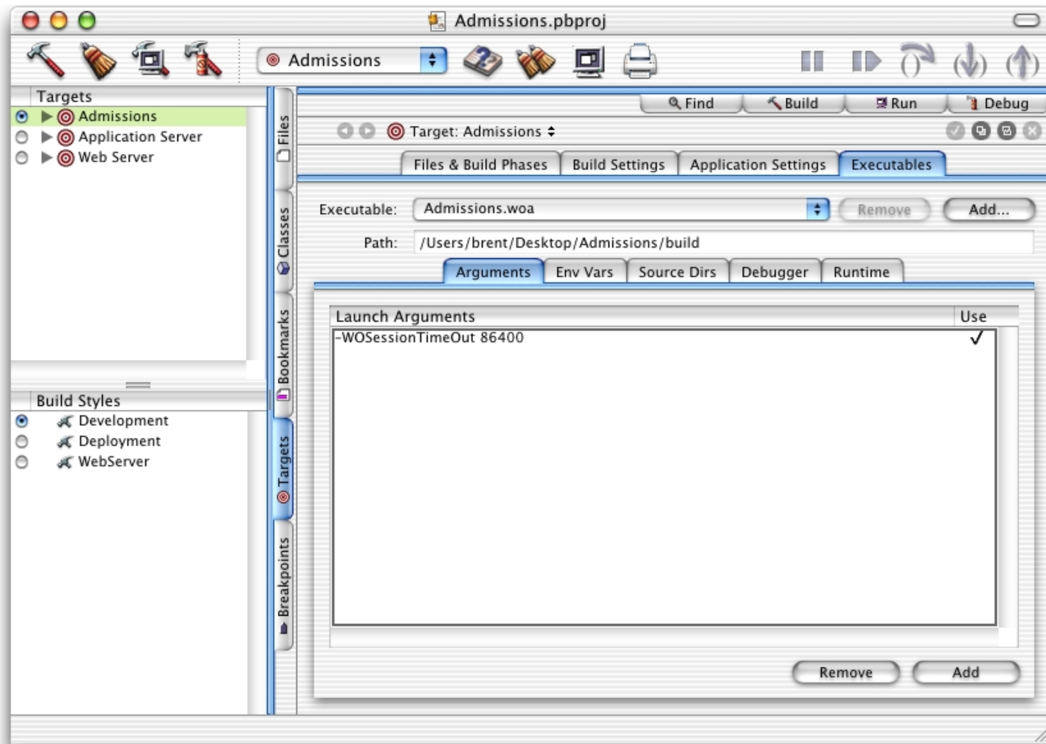
The default session timeout (60 minutes) is too short, so you need to set the timeout higher. Setting the timeout to 24 hours (86400 seconds) will better match the usage pattern of Java Client applications.

© Apple Computer, Inc. May 2002

Follow these steps to change the session timeout:

1. Click the Targets tab (one of the vertical tabs).
2. Select the Admissions target.
3. Click the Executables tab.
4. Click the Add button and enter `-WOSessionTimeOut 86400` as a launch argument as shown in Figure 3-14.

Figure 3-14 Add session timeout launch argument



© Apple Computer, Inc. May 2002

More About Session Time Outs

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 panel 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 `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's list as shown in Figure 3-14 (page 78).
2. Click the hammer icon in the toolbar to build the application. The Build pane slides down and displays all console messages during the build, including any errors.

© Apple Computer, Inc. May 2002

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 either of these techniques:

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

The book *Inside WebObjects: Deploying WebObjects Applications* covers the second and third options. You can run the server application from Project Builder by clicking the Launch icon or selecting 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. You can run it in a Web browser, start it from the command line, run it using Sun's `appletviewer`, 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, the client application is automatically started once the server application is up and running. So if you are developing with WebObjects 5.1 or later in Mac OS X, you can skip this section and continue with "Application Startup" (page 83).

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
```


© Apple Computer, Inc. May 2002

-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. This can be inconvenient during development, and setting a fixed port number using WOPort will make your life easier. 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.

On 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 Contents/MacOS. 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 path name from any directory in the shell: ~/Projects/Admissions/build/Admissions.woa/Contents/MacOS./Admissions_Client http://localhost:8888/cgi-bin/WebObjects/Admissions

© Apple Computer, Inc. May 2002

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. For developers new to UNIX, 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 WOJavaClientApplet component. 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
```

JDK appletviewer

The JDK's `appletviewer` is a useful tool during development because it allows you to test applets without needing to launch a Web browser. It downloads all the necessary classes automatically, just like a Web browser. The tool takes the URL of the server application as an argument, as shown here:

© Apple Computer, Inc. May 2002

```
[trivium] brent% appletviewer http://trivium.apple.com:8888/cgi-bin/
WebObjects/Admissions
```

MRJ Application

In Mac OS X, you can package Java Cilent applications as real double-clickable desktop applications. See the Java Client Launcher example in `/Developer/Examples/JavaWebObjects/JavaClientLauncher` for instructions.

Application Startup

It's important to understand how Java Client applications start up. With a few minor exceptions, applets and applications share the same startup process.

For both types of Java Client applications, the bindings in WOJavaClientApplet are sent to the client. If the client is an applet, the bindings are sent in HTML. If the client is an application, a WebObjects Direct Action creates an applet that sends the bindings to the client.

For applets, the main entry point is an instance of EOApplet that contains an `init()` method that accepts HTML arguments as parameters. For applications, the main entry point is an instance of EOApplication that contains a `main()` method that accepts command line arguments.

After one of the two entry point objects is created, the startup process is the same for both applets and applications. A method in EOApplication, `startApplication([args[])`, is invoked, and performs these operations:

1. If configured to do so, runs a URL dialog to ask for the application URL. This is necessary to get more information from the server, such as where the classes are located.
2. Instantiates the distribution channel (by default, this is `com.webobjects.eodistribution.client.EOHTTPChannel`) and establishes a connection to the server.

© Apple Computer, Inc. May 2002

3. For applications only, reads the arguments from the WOJavaClientApplet. This step doesn't happen for applets, since the arguments are read directly from the HTML file in which the applet is embedded.

4. Downloads classes to client (only if deployed as a desktop application).

5. Instantiates the application object, `com.webobjects.eoapplication.EOApplication`, for nondirect Java Client applications or `com.webobjects.eoapplication.EODynamicApplication` (or custom subclasses of) for Direct to Java Client applications. This step occurs after the classes are downloaded to the client.

6. Configures things such as user language and platform, performs some Swing initialization, including plugging into MRJ on Mac OS X, and loads user preferences.

7. Switches thread to continue execution in the main Swing event thread.

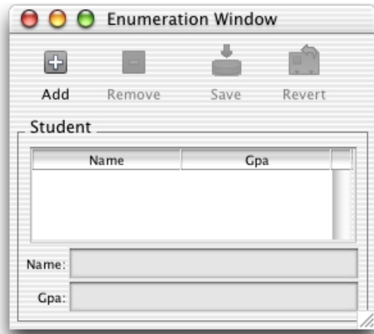
No user interface code or initialization should take place in these steps. After the last step, however, a method in EOApplication, `finishInitialization()` provides a place to initialize the interface controller (Java Client), warmup the controller factory (Direct to Java Client only), or perform other user interface-related operations.

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 3-15.

© Apple Computer, Inc. May 2002

Figure 3-15 Default enumeration window



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.

© Apple Computer, Inc. May 2002

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. Double-click `Admissions.eomodeld` which is in the Resources group.
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 3-16 (page 86).

Figure 3-16 Revised model

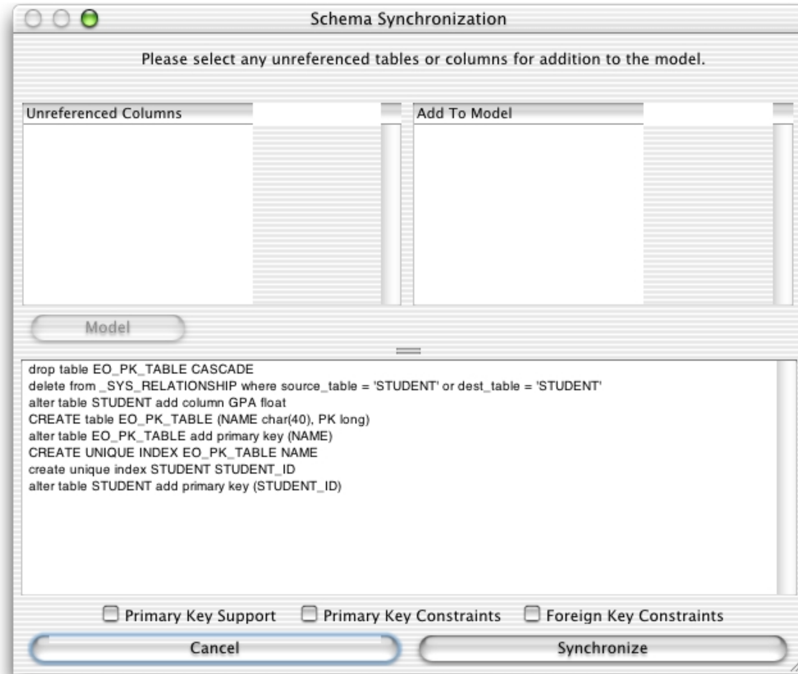
Student Attributes

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 3-17 (page 87).

© Apple Computer, Inc. May 2002

Figure 3-17 Schema Synchronization window



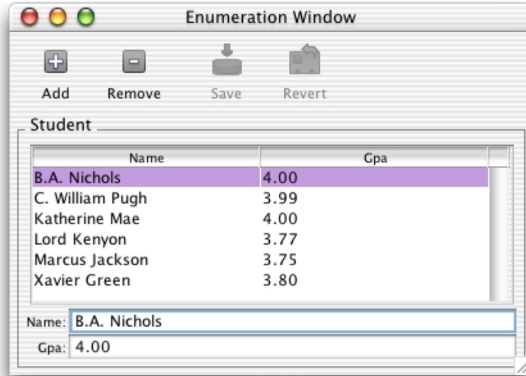
5. In OpenBase Manager, verify that the data type for the `gpa` attribute changed. Do this by choosing the Admissions database and clicking the Schema Design button to view the database's tables.

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 3-18 (page 88).

© Apple Computer, Inc. May 2002

Figure 3-18 Revised enumeration window



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

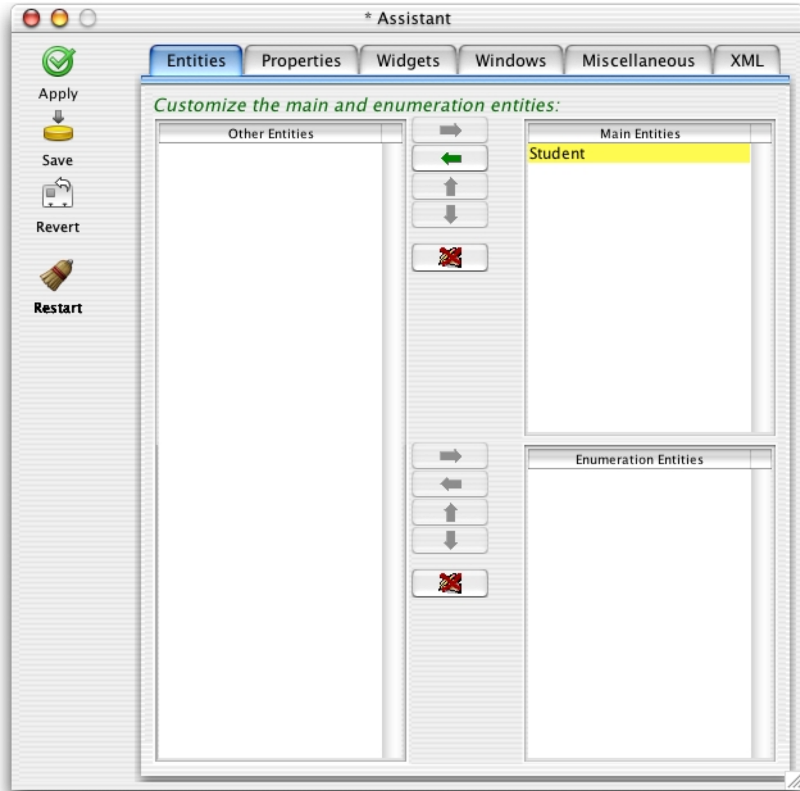
There are many ways to customize Direct to Java Client applications, including a tool called Assistant. Assistant is a Java application included in every Direct to Java Client application, and it provides an easy way to perform simple customizations. The following steps introduce you to Assistant:

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 3-19 (page 89).

© Apple Computer, Inc. May 2002

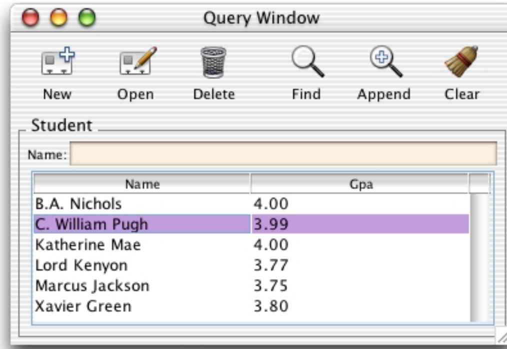
Figure 3-19 Change entity type



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 3-20 (page 90).

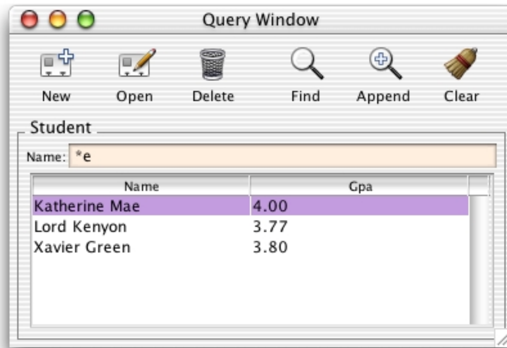
© Apple Computer, Inc. May 2002

Figure 3-20 Query window with data



4. Click New to add records; then enter different query strings to test the application. Figure 3-21 illustrates a query for names starting with "C."

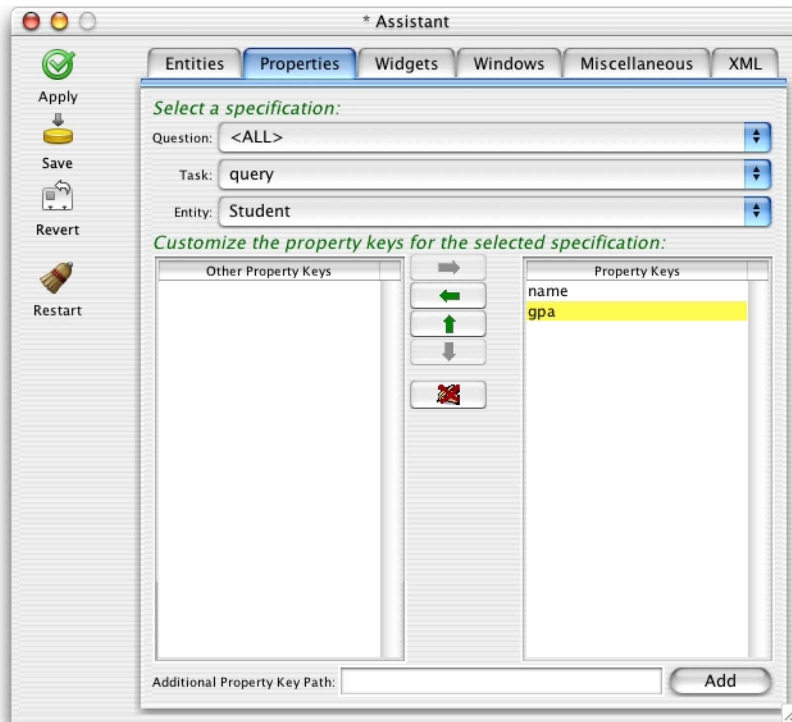
Figure 3-21 Query window searching for names containing "e"



© Apple Computer, Inc. May 2002

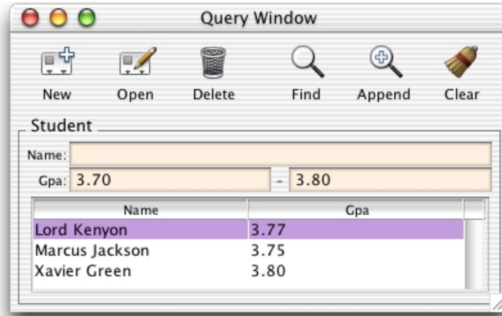
5. It would be nice to also query on the `gpa` attribute. In Assistant, switch to the Properties pane, and select "query" in the Task pop-up menu. You'll notice that the "gpa" property key is listed in the Other Property Keys list. As shown in Figure 3-22 (page 91), move it to the Property Keys list, save, and restart the application. You can now query on the `gpa` field also, as illustrated in Figure 3-23 (page 92). By default, the application provides two fields so you can search for a range of GPAs.

Figure 3-22 Properties tab in Assistant



© Apple Computer, Inc. May 2002

Figure 3-23 Query on GPA



6. You should probably beautify 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.

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.

Figure 3-24 (page 93) 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."

© Apple Computer, Inc. May 2002

Figure 3-24 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 3-25 (page 93) 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.

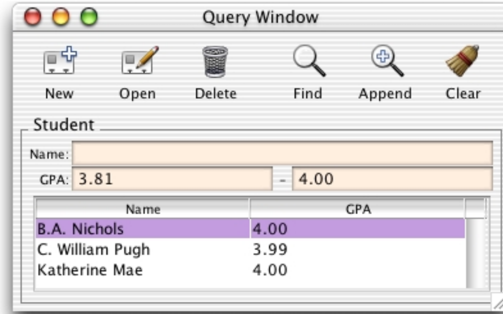
Figure 3-25 Right-hand side of rules

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 3-26 (page 94).

© Apple Computer, Inc. May 2002

Figure 3-26 The application with simple customizations



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.

© Apple Computer, Inc. May 2002

Inside Assistant

This section gives more details about Assistant, the first tool for customizing Direct to Java Client Applications.

Entities

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

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 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

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

© Apple Computer, Inc. May 2002

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 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 simply other entities. Other entities can be manipulated through the master-detail user interfaces of main entities.

Properties

The Properties pane lets you see how the EOGeneration layer interpreted the entities and their attributes in your EOModel.

Task

The Task pop-up menu identifies the properties that are displayed for each entity. That is, you can choose which properties (attributes) are displayed in forms, modal dialogs, query windows, and data lists. The different tasks correspond to different window types:

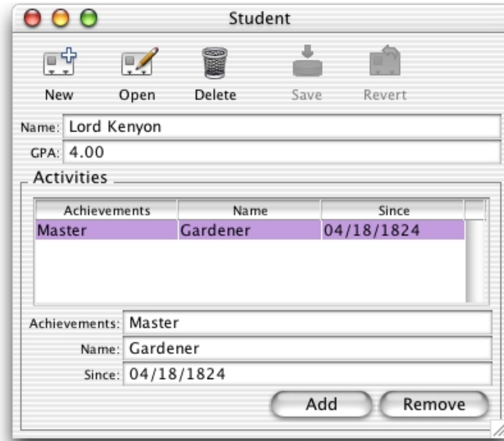
form

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

Figure 3-27 (page 97) shows a form window.

© Apple Computer, Inc. May 2002

Figure 3-27 Form window



identify

Used whenever an object is simply referenced in the user interface, such as in a master-detail interface or an error dialog.

list

Displays the results of a query in a table view for attributes that are main property keys.

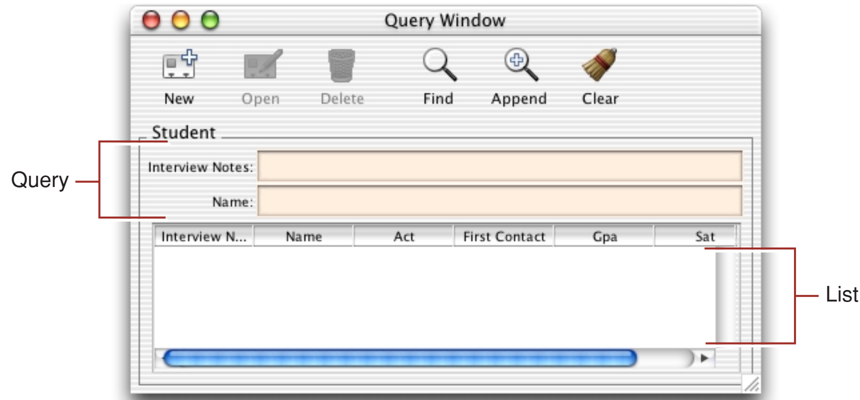
query

Provides a text field to query on for each property key.

Figure 3-28 (page 98) shows a query window.

© Apple Computer, Inc. May 2002

Figure 3-28 Query window and list task



Question

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, select it in the Question menu before making changes to the task and property keys.

Property Keys

The property keys displayed in Assistant are all the attributes of an entity that are client-side class properties.

Additional Property Key Path

You can add property keys for methods in your business logic classes using this field.

© Apple Computer, Inc. May 2002

Widgets

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

Windows

You use the options in this pane to change the controller class for windows in your application, such as the title aspect of query and enumeration windows.

Miscellaneous

This pane contains some additional options for tweaking widgets.

XML

You use the information in this pane when freezing XML files. It contains the XML description of the dynamically generated user interface. The Save button creates a text file of the XML description.

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:

© Apple Computer, Inc. May 2002

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. Add a foreign key by copying Student’s primary key (`studentID`) into the Activity table. In diagram view, 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 or as a server-side class property in the Activity entity.
4. Make `activityID` the primary key in the Activity entity by clicking in the key column. The new entity should look as shown in Figure 3-29.

Figure 3-29 Activity entity

Activity Attributes

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	

© Apple Computer, Inc. May 2002

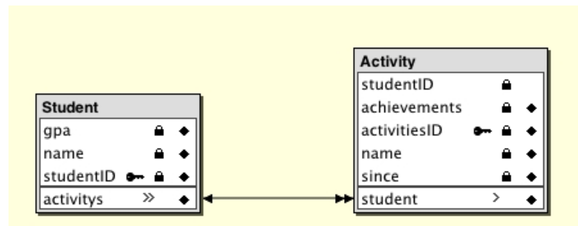
5. Select the Activity entity in the entities list and choose Generate SQL from the Tools menu. Since you already generated primary key support the first time you generated SQL, make sure to deselect the option Create Primary Key Support. Primary Key Constraints should be selected so Activity's primary key is correctly marked in the database. Don't select Foreign Key Constraints.

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 3-30.

Figure 3-30 Relate Student and Activity



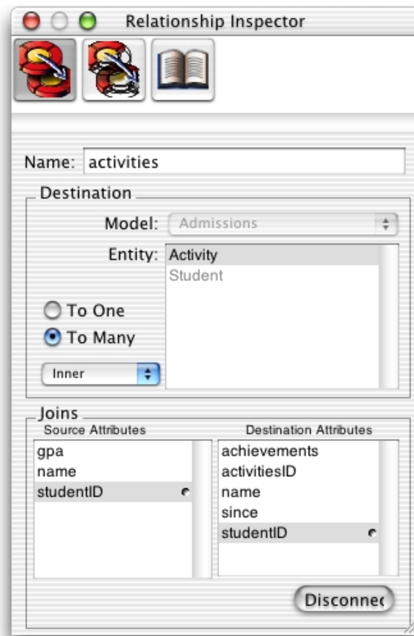
   This action creates a relationship in both entities: a to-many relationship from Student to Activity and a to-one relationship from Activity to Student.

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

© Apple Computer, Inc. May 2002

Change the relationship name to “activities.”

Figure 3-31 Relationship Inspector for Student’s `activities` relationship

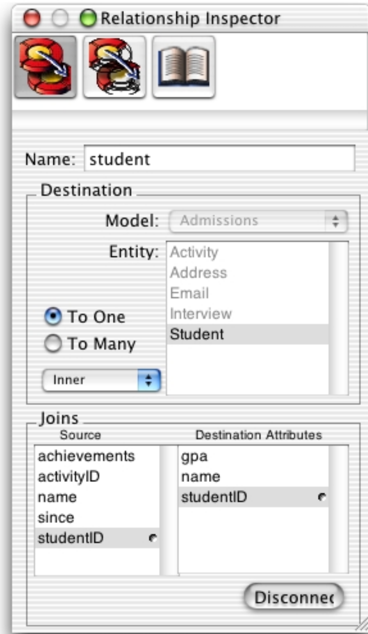


3. Since each student can have multiple activities, the relationship from the Student entity to the Activity entity is a to-many relationship. Select the Student entity and make sure To Many is selected and that `studentID` is selected in both the Source Attributes list and the Destination Attributes list, as shown in Figure 3-31.

4. Each activity and its attributes are unique to a single student, so the relationship from Activity to Student is a to-one relationship. Select the Activity entity and make sure To One is selected. Also verify that `studentID` is selected in both attributes lists as shown in Figure 3-32 (page 103).

© Apple Computer, Inc. May 2002

Figure 3-32 Relationship Inspector for Activity's `student` relationship



5. As with entity attributes, you can choose to pass relationships to the client. You need to add the new relationships as client-side class properties. Switch to table mode (Tools > Table Mode). Below the attributes pane is a pane 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 entity list, its relationship (`activities`) is shown in this pane, as Figure 3-33 (page 104) illustrates. Selecting the Activity entity displays its relationship (`student`), as Figure 3-34 (page 104) 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 3-33 (page 104). However, do not make the `student` relationship in the Activity entity a client-side class property, as shown in Figure 3-34 (page 104).

© Apple Computer, Inc. May 2002

Figure 3-33 Make Student to Activity relationship a client-side class property

Student Relationships

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

Figure 3-34 Do not make Activity to Student relationship a client-side class property

Activity Relationships

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

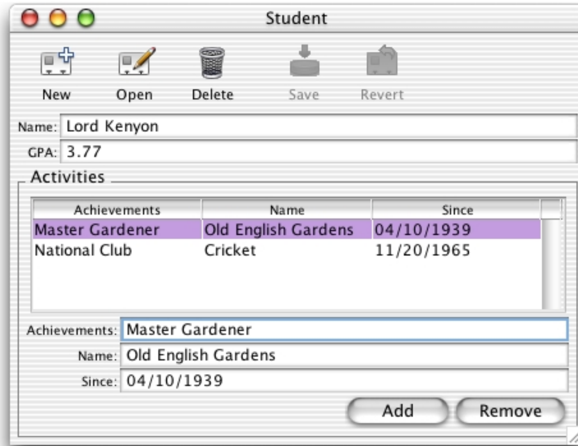
You do not need to synchronize the schema as the Enterprise Object technology manages the relationships for you. This helps you build reusable enterprise object models since the relationship is not database-specific.

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 3-35 (page 105).

© Apple Computer, Inc. May 2002

Figure 3-35 Add activities to new Student record



The rule system considers the Activity entity to be of the entity type "other." ""Other" Entities" (page 96) describes why, when analyzing the enterprise object model containing the Activity entity, the rule system considers Activity an "other" entity. This allows you to add activities to a Student record by clicking the Add button in form windows for the Student entity.

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.

© Apple Computer, Inc. May 2002

Where to Go From Here

Chapter 4, "Distribution Layer" (page 107), 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 Chapter 5, "Advanced Tutorial." 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 Chapter 5 (page 119).

© Apple Computer, Inc. May 2002

CHAPTER 4

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 behaving, 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 intra-layer 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.

© Apple Computer, Inc. May 2002

In Java Client, however, business objects are sent to the client application. This is done for performance reasons. Java Client applications generally access more data than do distributed HTML applications, and 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, and Java objects can quite easily be decompiled and analyzed. 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 at the same time 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:

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


© Apple Computer, Inc. May 2002

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

```
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:

```
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 128) 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 instance, if a particular computation requires a lot of data, and the client does not already have the data, it makes sense for that computation to occur on the server, since the server is closer to the data store.

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

© Apple Computer, Inc. May 2002

Remote Method Invocations

In Java Client applications you may want some methods to execute only on the server. This is particularly the case when security is an issue, but performance can be a reason as well (as when the method consumes a lot of system resources). Java Client defines two categories of remote method invocations: those that apply to business logic and those that apply to application logic.

On 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: 1) the method to invoke in the server-side class; 2) a `java.lang.Class` object representing the argument types; 3) 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.

© Apple Computer, Inc. May 2002

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 will 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 117).

On 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 WOJavaClientApplet object.

© Apple Computer, Inc. May 2002

- If `keyPath` is an empty string, the method is invoked on the WOComponent that is the invocation target of the EODistributionContext (typically a subclass of WOJavaClientApplet).
- 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. That is, if an actual key path is specified, the EODistributionContext on the server blocks all invocations sent with this method unless the `methodName` argument is prefixed with `clientSideRequest` or unless the EODistributionContext's delegate (on the server) implements `distributionContextShouldAllowInvocation` and `distributionContextShouldFollowKeyPath`. For security reasons, the delegate must authorize the invocation and the key path in these methods.

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."

© Apple Computer, Inc. May 2002

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`.

`awakeFromClientUpdate` 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 a guaranteed to have all the business data from the client.

`prepareValuesForClient` 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.

© Apple Computer, Inc. May 2002

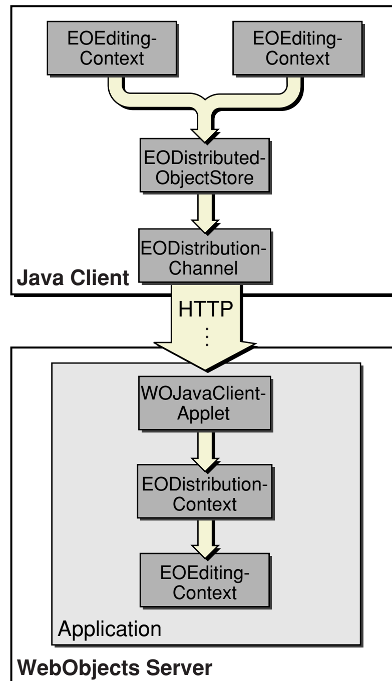
Distribution Layer Objects

The distribution layer's client-server communication mechanism relies on four objects: `com.webobjects.eodistribution.WOJavaClientApplet`, `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 a EODistributionChannel to transfer objects across the network to the WOJavaClientApplet, 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 4-1.

© Apple Computer, Inc. May 2002

Figure 4-1 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 WOJavaClientApplet 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 either deployment environment, and it also plays a large role in application startup. See “Application Startup” (page 83) for more information on this object. It is also the object which embeds Java Client in a WebObjects application.

© Apple Computer, Inc. May 2002

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:

- The user makes a query and the fetch specification is forwarded by the client's EODistribution layer to the server's EODistribution layer.
- The normal WebObjects mechanisms take over, and a SQL call is eventually made to the database server.
- The database server returns rows of requested data that is mapped to enterprise objects.
- The server's EODistribution layer sends copies of the requested data to the client.
- 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).
- 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.

© Apple Computer, Inc. May 2002

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.

Distribution Channels

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.

See the EODistribution API reference for more details.

Delegates

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

© Apple Computer, Inc. May 2002

You set custom delegates with the `setDelegate` method in EODistributionChannel and EODistributionContext. If you use custom encyrption and decryption, you must be aware of the timing issues involved. To be effective, the delegates for doing these things must be in place before the first byte of data is transferred, which requires you to subclass EOApplication. You then must add code on the server-side to get the timing right, like registering for the `EODistributionContextInstantiatedNotification`.

Refer to the EODistribution API reference for detailed information.

© Apple Computer, Inc. May 2002

CHAPTER 5

Advanced Tutorial

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.

The first customization tool is the Direct to Java Client **Assistant**, which you've already used in Chapter 3, "Basic Tutorial." 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

© Apple Computer, Inc. May 2002

- 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.) For this reason, 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 Chapter 7, “Inside the Rule System.”

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.

© Apple Computer, Inc. May 2002

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 "Task: Freezing XML User Interfaces" (page 227).

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. Chapter 14, "Task: Mixing Static and Dynamic User Interfaces" (page 241), 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.

© Apple Computer, Inc. May 2002

Table 5-1 compares the five customization techniques using several criteria.

Table 5-1 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

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.

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

© Apple Computer, Inc. May 2002

- 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.

Since you added attributes to the entity, you must synchronize the model and the database schema that generates the appropriate SQL for the updated entity. Refer to "Using the Application" (page 84) and Figure 3-17 (page 87) for a reminder.

The Student entity should now resemble Figure 5-1.

Figure 5-1 The updated Student entity

Student Attributes

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 an Entity

Each student record can be associated with one or many interviews, so you need a new entity to hold the interview records. In EOModeler, complete the following steps to enhance the model:

1. Add a new entity named "Interview" with table name "INTERVIEW." Its class is EOGenericRecord.
2. Add attributes to the new entity:
    - Name: `interviewID`; Column: INTERVIEW_ID; External Type: `int`; Internal Data Type: `Integer`. Do not make this a client-side class property or a server-side class property.

© Apple Computer, Inc. May 2002

- Name: `interviewDate`; Column: INTERVIEW_DATE; External Type: `datetime`; Internal Data Type: `Date`. Make this a client-side class property. Verify that it is also marked as a server-side class property. Don't lock on this attribute: Deselect the lock icon in the attribute's row.
- Name: `interviewNotes`; Column: INTERVIEW_NOTES; External Type: `char`; Internal Data Type: `String`, width 4096. Make this a client-side class property. Verify that it is also marked as a server-side class property.

3. Add a foreign key to the entity by copying Student's primary key (`studentID`) in the Student table, choose Copy from the Edit menu, click in the Interview table, and choose Paste from the Edit menu. Verify that `Interview.studentID` is not marked as a primary key or as a server-side class property in the Activity entity.
4. Make `interviewID` the primary key in the Interview entity by clicking the key field. The new entity should look as shown in Figure 5-2 (page 124).

Figure 5-2 Interview entity

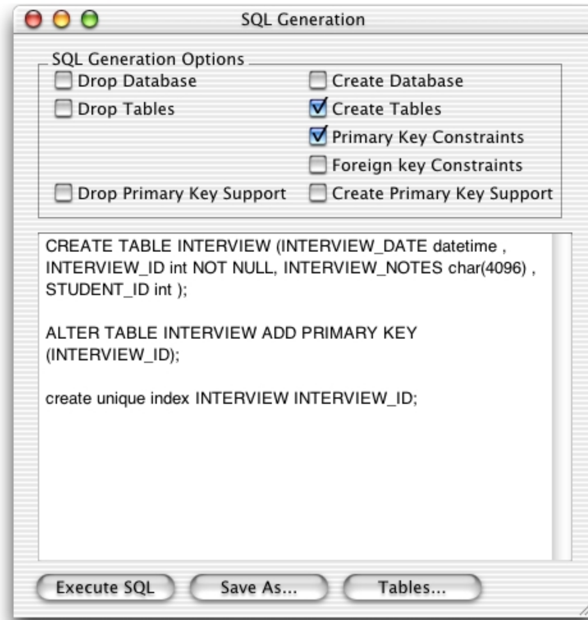
Interview Attributes

Name	Column	Value Class (Java)	External Type	Width
interviewDate	INTERVIEW_DATE	NSTimestamp	datetime	
interviewID	INTERVIEW_ID	Number	int	
interviewNotes	INTERVIEW_NOTES	String	char	4096
studentID	STUDENT_ID	Number	int	

5. Select the Interview entity and choose Generate SQL from the Tools menu. Since you already generated primary key support the first time you generated SQL, make sure to deselect the option Create Primary Key Support. As shown in Figure 5-3 (page 125), Primary Key Constraints should be selected so Interview's primary key is correctly marked in the database. Don't select Foreign Key Constraints.

© Apple Computer, Inc. May 2002

Figure 5-3 Generate SQL for the Interview entity



Make a Relationship

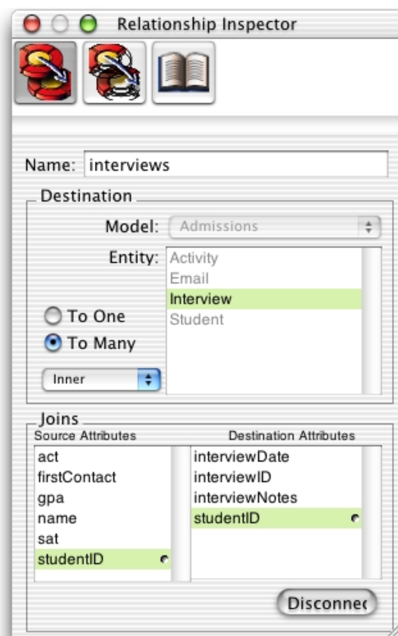
Follow these steps to relate Student and Interview in a to-many relationship:

1. In a diagram view (Tools > Diagram View), Control-drag from Student's primary key (`studentID`) to Interview's foreign key (`studentID`). This action creates a relationship in both entities: a to-many relationship from Student to Interview and a to-one relationship from Interview to Student.
2. The Relationship Inspector allows you to customize the relationship. In table mode, select the Student entity and then select its `interviews` relationship. Then, if it is not visible on the screen, select Inspector from the Tools menu. Verify that the relationship to Interview in the Student entity is named "interviews."

© Apple Computer, Inc. May 2002

3. Since each student can have multiple interviews, the relationship from Student to Interview is a to-many relationship. To-many relationships join on the origin's primary key and on the destination's foreign key. Select the Student entity in the Relationship Inspector and verify that To Many is selected and that `studentID` is selected in both the Source Attributes list and the Destination Attributes list as shown in Figure 5-4 (page 126).

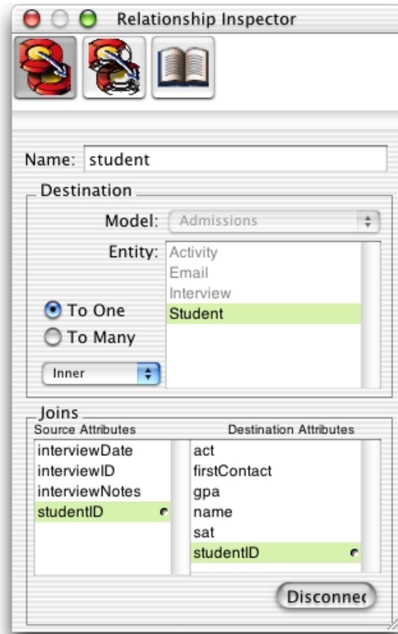
Figure 5-4 The `interviews` relationship in the Student entity



4. Each interview is specific to a single student, so the relationship from Interview to Student is a to-one relationship. The inverse relationship of a bidirectional to-many relationship joins on the origin's foreign key (`Interview.studentID`) and on the destination's primary key (`Student.studentID`). To verify this, first select the Interview entity in table mode. Then in the Relationship Inspector, verify that To One is selected and that `studentID` is selected in both the Source Attributes list and the Destination Attributes list.

© Apple Computer, Inc. May 2002

Figure 5-5 The `student` relationship in the Interview entity



5. As with attributes in entities, you can choose to pass relationships to the client. You need to add the new relationships as client-side class properties. Just as you did with the `activities` relationship for the Student entity in the basic tutorial, make the `interviews` relationship in the Student entity a client-side class property. However, don't make the inverse relationship (the student relationship in the Interview entity) a client-side class property. See "Make the Relationship" (page 101) for a refresher and refer to Figure 5-6 (page 127) and Figure 5-7 (page 128) to see how the final result should look.

Figure 5-6 Student's relationships

Student Relationships

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

© Apple Computer, Inc. May 2002

Figure 5-7 Interview's relationship

Interview Relationships

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

Add Custom Business Logic

As the basic tutorial illustrates, you can go far in 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 your enterprise objects brings your 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`.

© Apple Computer, Inc. May 2002

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 108) 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, however, you 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:

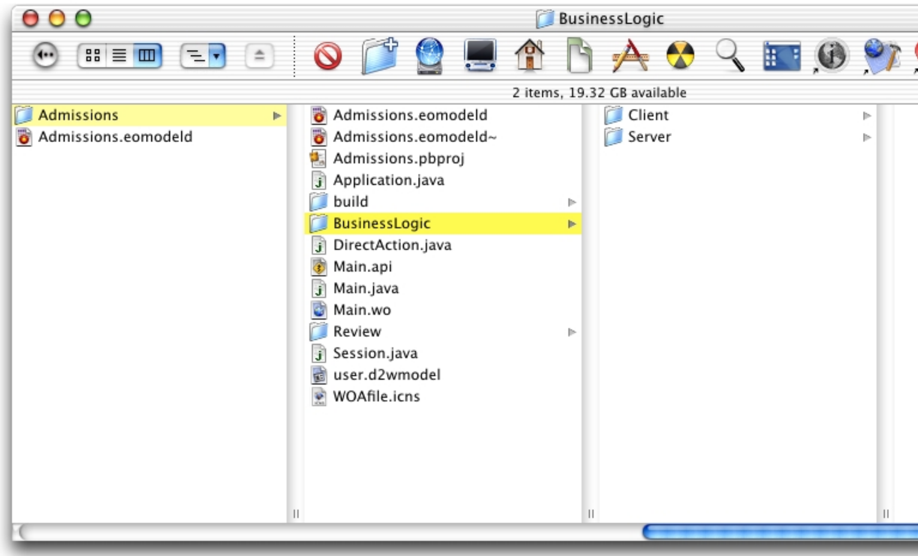
Create the following directories at the top level of your project directory (do this in the file system, not in Project Builder):

```
BusinessLogic
BusinessLogic/Client
BusinessLogic/Server
```

The directory structure should look like Figure 5-8.

© Apple Computer, Inc. May 2002

Figure 5-8 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.

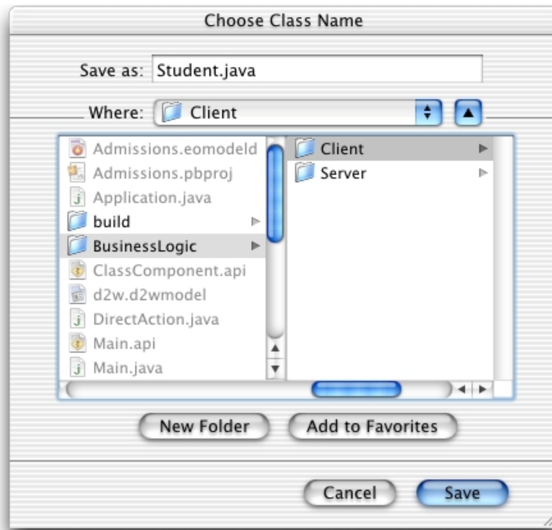
Follow these steps to generate Java files for the client:

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 5-9.

© Apple Computer, Inc. May 2002

4. Click Save.
5. Repeat the process for the Activity entity.

Figure 5-9 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.

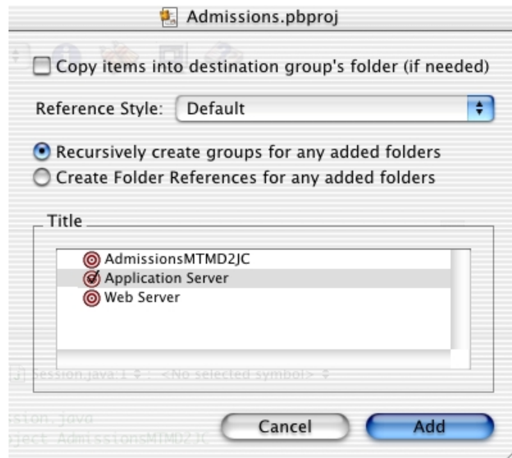
© Apple Computer, Inc. May 2002

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 Groups & Files pane of Project Builder.
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 5-10 (page 132). Also make sure that "Recursively create groups for any added folders" is selected.

Figure 5-10 Import BusinessLogic directory



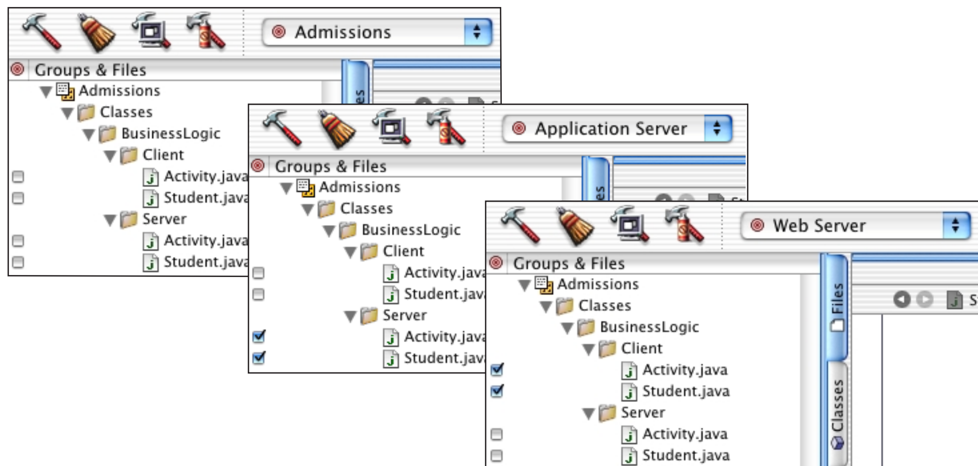
5. Click Add. The new files should appear in the Groups & Files pane as illustrated in Figure 5-11.
6. After the import, change the target for the files in BusinessLogic/Client to Web Server. Make sure you also disassociate the files in BusinessLogic/Client from the Application Server target by switching to that target and deselecting the

© Apple Computer, Inc. May 2002

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 5-11 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 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.

© Apple Computer, Inc. May 2002

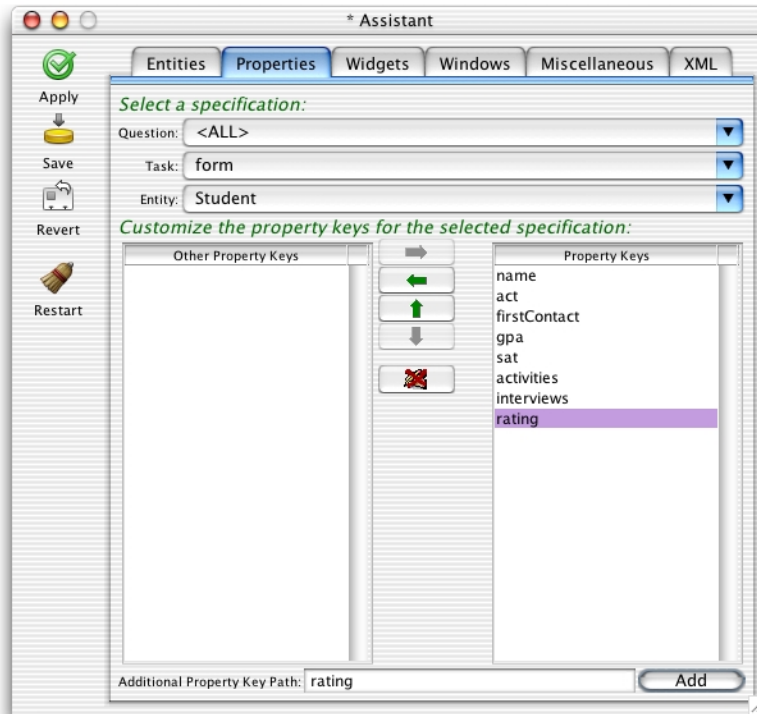
Prepare Application for Business Logic

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, Entity=Student using the Additional Property Key Path text field and the Add button as shown in Figure 5-12.

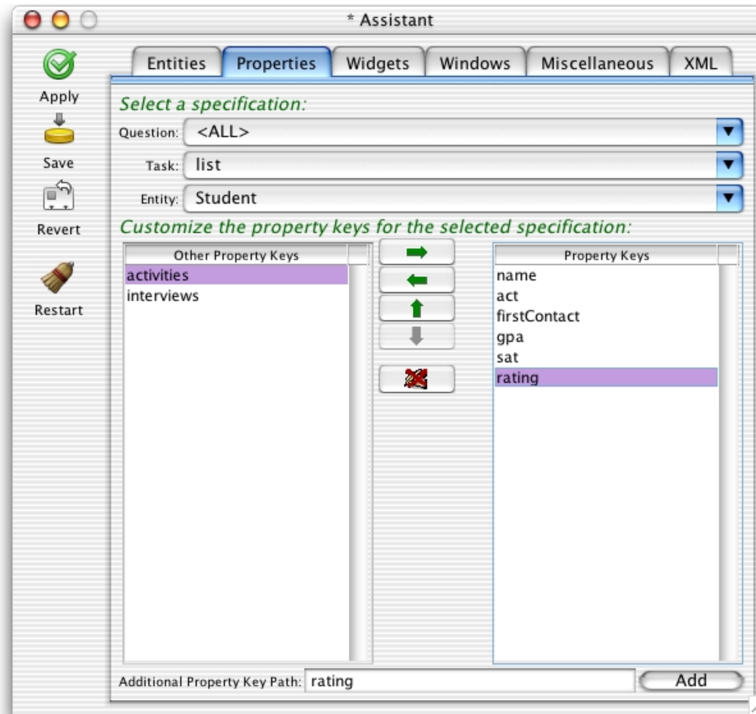
Figure 5-12 Add a property key for the form task



© Apple Computer, Inc. May 2002

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. Switch Task to list and click Add, as shown in Figure 5-13.

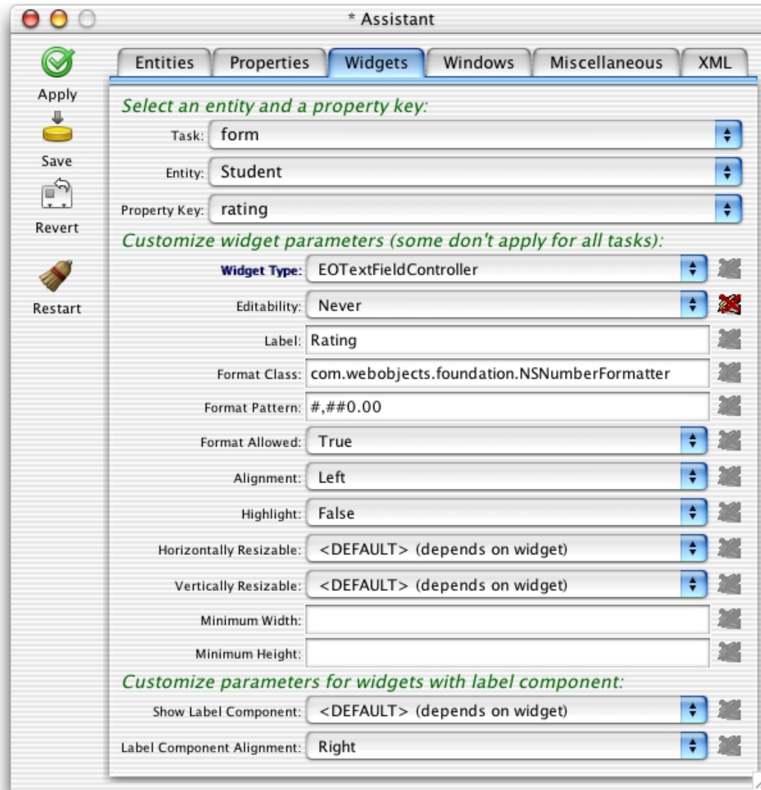
Figure 5-13 Additional property key for list task



4. The new property will be associated (via an EOAssociation, see "Associations" (page 48)) 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 select Task=form, Entity=Student, Property Key=rating. From the Widget Type pop-up menu, select EOTextFieldController if it is not already selected. Doing this binds the association aspect of the EOTextFieldController widget (rating) with the `rating` method, which you'll define in a few steps.

© Apple Computer, Inc. May 2002

Figure 5-14 Change the widget type to make the association.



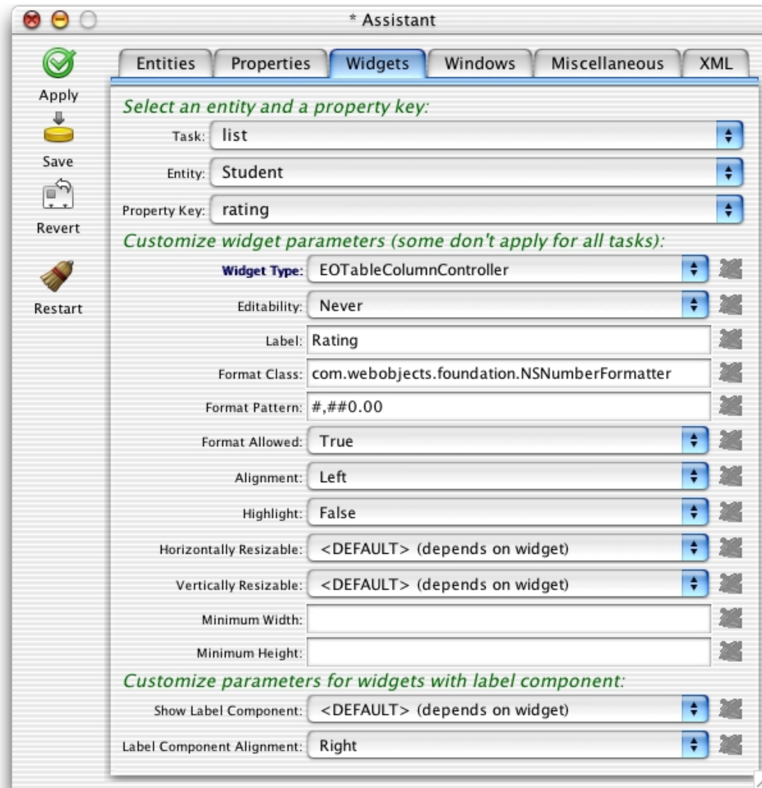
5. Since the rating is calculated on the server side, the text field should be marked as not editable by the user. So, while in the Widgets pane, select Never in the Editability pop-up menu.

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 5-14. See the class reference documentation for NSNumberFormatter for more information on format patterns.

© Apple Computer, Inc. May 2002

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 5-15.

Figure 5-15 Change formatter for property in list view



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

© Apple Computer, Inc. May 2002

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));
 }
```


© Apple Computer, Inc. May 2002

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.

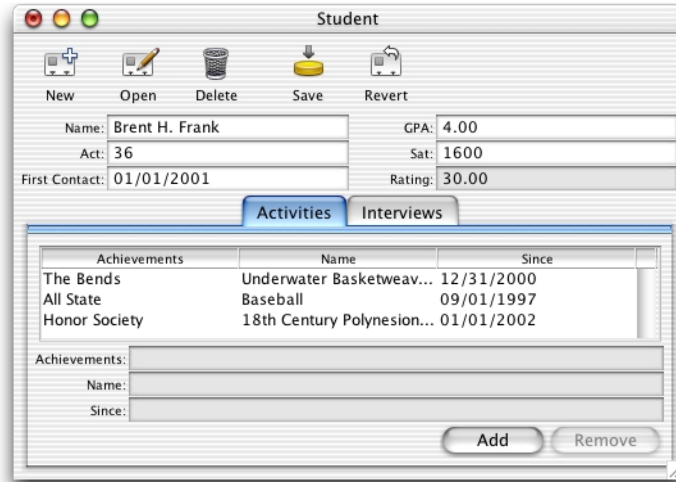
There is more going on behind the scenes, though. The `rating` in the client-side business logic class invokes a remote method called `clientSideRequestRating` in the server-side business logic class. This method in turn invokes a method called `rating`, which actually performs the calculation.

Rebuild and run the application. Make a new student record and see how the rating field is populated upon saving as shown in Figure 5-16 (page 140).

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.

© Apple Computer, Inc. May 2002

Figure 5-16 The rating field in action



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;
}
```


© Apple Computer, Inc. May 2002

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

© Apple Computer, Inc. May 2002

Initial Values

When you create a new record, it would be nice 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.

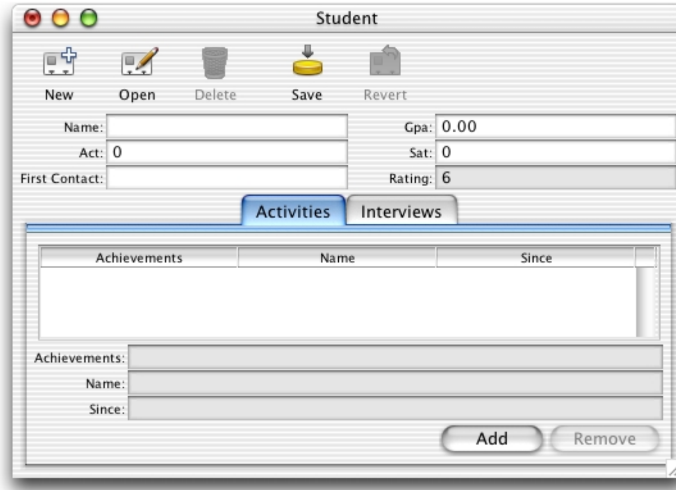
Add this code in the server-side Student.java file:

```
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 5-17 (page 143).

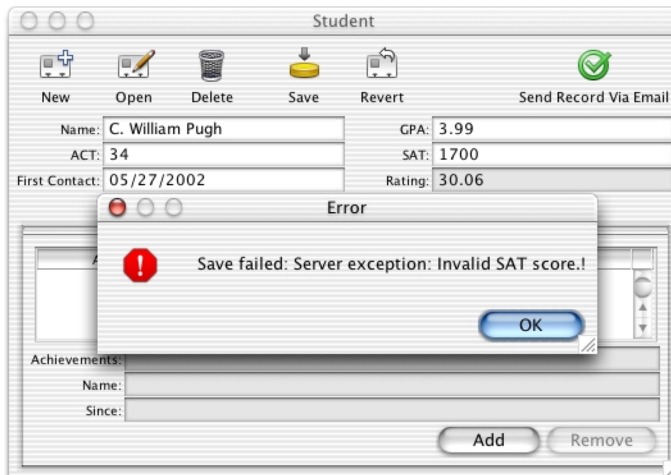
© Apple Computer, Inc. May 2002

Figure 5-17 Initial values



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 5-18 (page 143).

Figure 5-18 Validation exception message



© Apple Computer, Inc. May 2002

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 check-boxes.

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.

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.

© Apple Computer, Inc. May 2002

- **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.client.EOControllerFactory` object, the rule system, and D2WComponent objects.

An EOControllerFactory is created on the server during the server application's initialization, and this object creates the controller hierarchy. It does this using the rule system, which provides XML descriptions of controller hierarchies. The controller factory then parses the XML (using a `com.webobjects.eoapplication.EOXMLUnarchiver` object) and generates the specified controllers.

The EOXMLUnarchiver maps XML tags to EOController classes, as illustrated in Table 5-2.

Table 5-2 A subset of the controllers available in Direct to Java Client

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

© Apple Computer, Inc. May 2002

Table 5-2 A subset of the controllers available in Direct to Java Client (continued)

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.

For more information on the XML tags and attributes for controller classes, see Appendix A (page 297).

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.

© Apple Computer, Inc. May 2002

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 Appendix A (page 297).

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 "XML Description of Classes and Actions" (page 297).

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` 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 your database's performance and in turn your application's usability as users are more likely to invoke unqualified searches.

© Apple Computer, Inc. May 2002

See "Inside the Rule System" (page 189) for an explanation of rule priorities and for more general information on the rule system. Also see "Task: Customizing With Common Rules" (page 221) 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; action method is in client-side business logic)
- subclass a controller class and write a rule to use it in the application (action method is in subclass)
- write a custom controller class and include it in an XML description (for actions on user interface; action method is in custom controller class)
- edit XML by hand to include an EOActionController with an `actionKey` tag specifying the action method (for actions on enterprise objects; action method is in client-side business logic)

Write the Action

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.

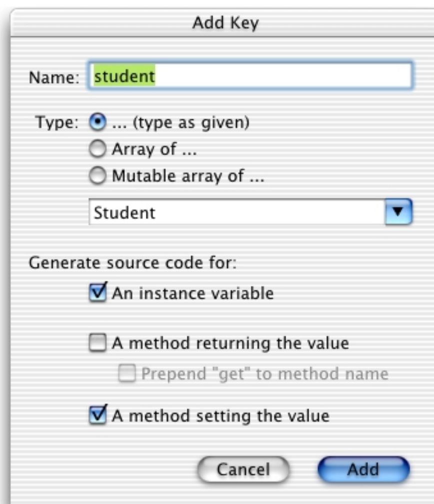
Note: You'll better understand this part of the tutorial if you're familiar with the concepts involved in an HTML WebObjects application. The book *Inside WebObjects: Discovering WebObjects for HTML* is a great place to start learning.

© Apple Computer, Inc. May 2002

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 5-19. Select the checkboxes to generate source code for an instance variable and a method setting the value.

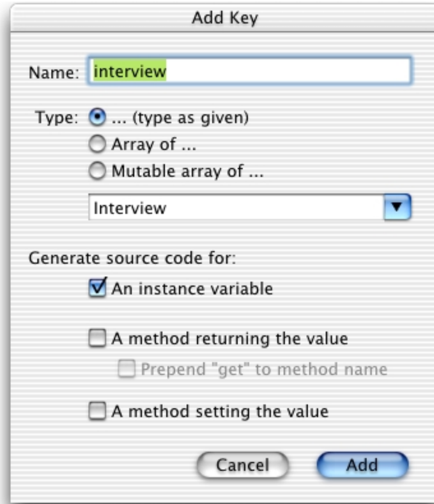
Figure 5-19 New key of type Student in the Report component



3. Add another new key called "interview" of type Interview, as shown in Figure 5-20. Select the checkbox to generate source code for an instance variable.

© Apple Computer, Inc. May 2002

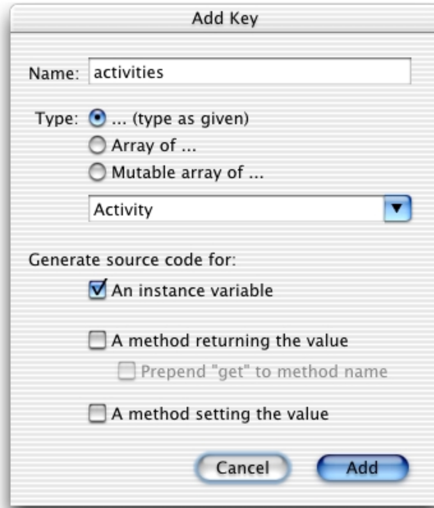
Figure 5-20 New key of type Interview in the Report component



4. Add another new key called "activities" of type Activity, as shown in Figure 5-21. Select the checkbox to generate source code for an instance variable.

© Apple Computer, Inc. May 2002

Figure 5-21 New key of type Activity in the Report component



5. Add dynamic elements for Student's attributes. Add WOStrings for the `gpa`, `act`, `sat`, and `name` attributes as shown in Figure 5-22. They are shown here in a table, but that is optional.

Figure 5-22 Dynamic elements for Student's attributes

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

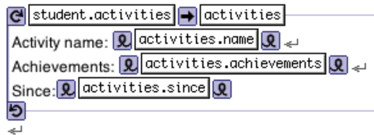
6. Add dynamic elements for Student's `interviews` relationship. Add a WORepetition with `list = student.interviews` and `item = interview`. Add a WOString for `interview.interviewDate` and a WOString for `interview.interviewNotes` within the repetition as shown in Figure 5-23.

© Apple Computer, Inc. May 2002

Figure 5-23 WORepetition for Student's interviews

7. Add dynamic elements for Student's `activities` relationship. Add a WORepetition with `list=student.activities` and `item = activities`. Add WOStrings for `activities.name`, `activities.achievements`, and `activities.since` as shown in Figure 5-24.

Figure 5-24 WORepetition for Student's activities



8. 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);
}
```


© Apple Computer, Inc. May 2002

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.

9. 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. 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"));
}
```

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.

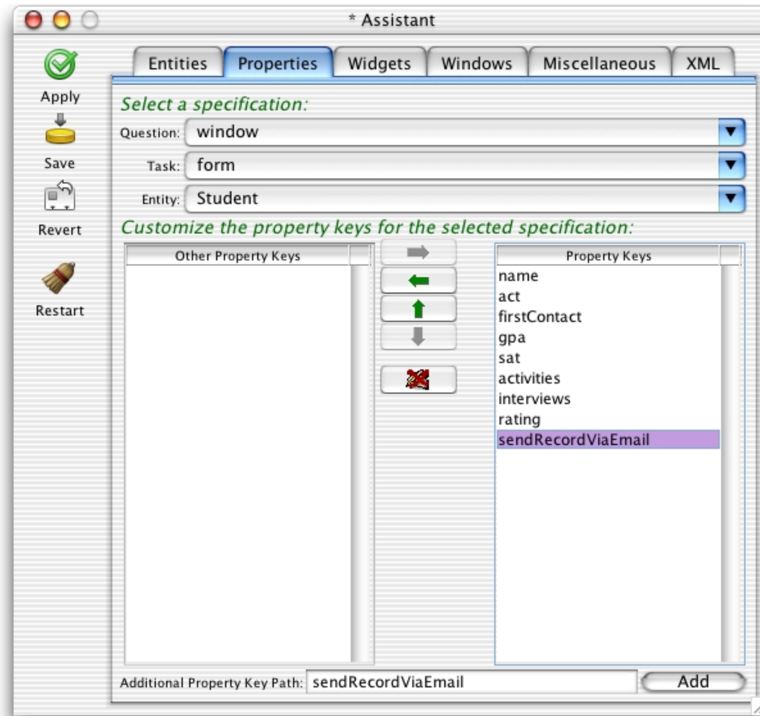
Use Assistant

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

1. Build and run the Admissions application and open Assistant.
2. Switch to the Properties pane and 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 5-25.

© Apple Computer, Inc. May 2002

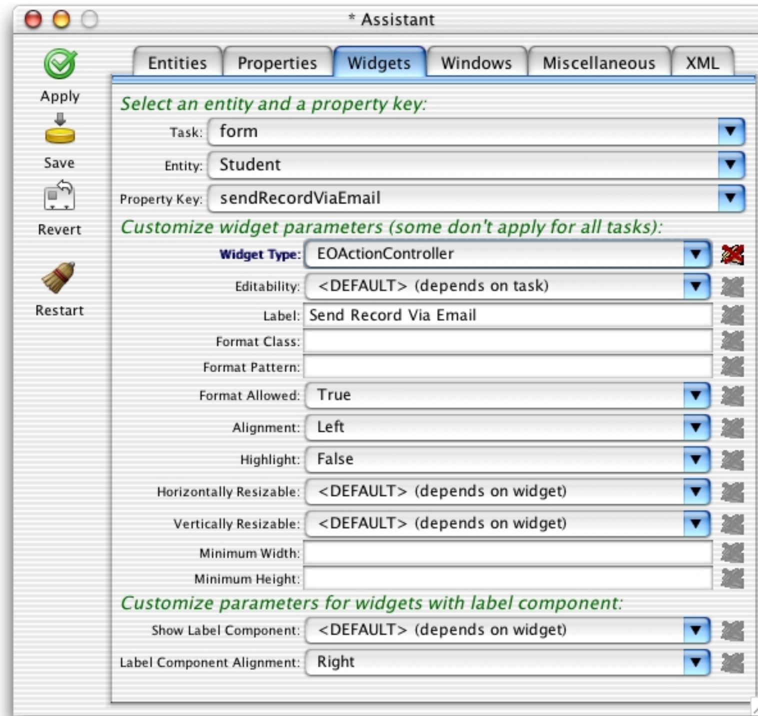
Figure 5-25 Add property key for new action



3. Switch to the Widgets pane, select Task=form, Entity=Student, and Property key=sendRecordViaEmail. In the Widget Type pop-up menu, select EOActionController as shown in Figure 5-26.

© Apple Computer, Inc. May 2002

Figure 5-26 Change the widget type of the new property key



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 5-27. 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).

© Apple Computer, Inc. May 2002

Figure 5-27 The new property key as an EOActionController

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.

© Apple Computer, Inc. May 2002

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 chapter Chapter 4, "Distribution Layer" (page 107),for an introduction to the distribution layer and remote method invocation.

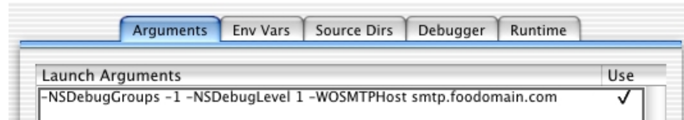
Next, you need to add the import statement for the client-side EODistribution layer to the `Student.java` class:

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

Finally, 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 5-28 (page 158). Refer to "Add a Launch Argument" (page 77) if you've forgotten how to add a launch argument.

© Apple Computer, Inc. May 2002

Figure 5-28 Add launch argument for SMTP host



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 in box 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; 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 5-1.

© Apple Computer, Inc. May 2002

Listing 5-1 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.client.*;
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);
    }
```


© Apple Computer, Inc. May 2002

```
    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:

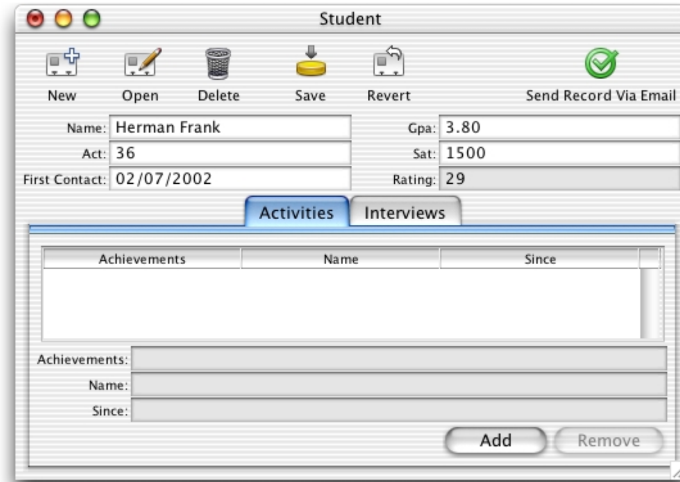
Left-Hand Side: `((task='form') and (controllerType='entityController') and (entity.name='Student'))`
Key: `className`
Value: `"admissions.client.CustomFormController"`
Priority: `50`

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.

Build and run the application and remove the action you added with Assistant (you can either make this an "Other Property Key" in Assistant 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 5-29.

© Apple Computer, Inc. May 2002

Figure 5-29 Image form window with new buttons



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, in respect of 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 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) are controller classes and should not include any user interface code. So, add a new client-side class to your project called SelectEmail:

© Apple Computer, Inc. May 2002

```
package admissions.client;

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

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 5-30 (page 164).

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. The Email entity is considered an Enumeration entity by the rule system, so you can add data to it by selecting 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();
```


© Apple Computer, Inc. May 2002

```
        _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:

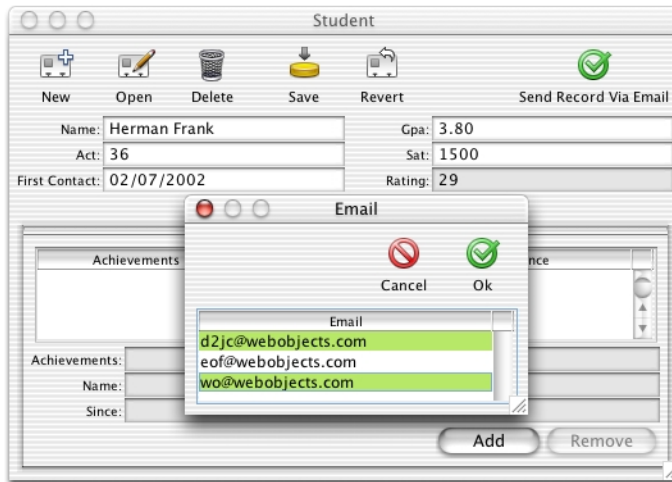
```
public void clientSideRequestSendRecordViaEmail(EOEnterpriseObject record, NSArray
                                                                       sendTo) {
        String messageSubject, messageBody, message;
        NSMutableArray recipients = new NSMutableArray();
        //recipients.addObject("person@foo.com");

        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);
}
```


© Apple Computer, Inc. May 2002

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 the that shown in Figure 5-30 should appear. Select some email addresses and click Ok. Check your email to see if you are successful.

Figure 5-30 Choose email recipients



© Apple Computer, Inc. May 2002

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 Javadoc API reference for NSLog for more information on how to use NSLog.

© Apple Computer, Inc. May 2002

© Apple Computer, Inc. May 2002

C H A P T E R 6

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

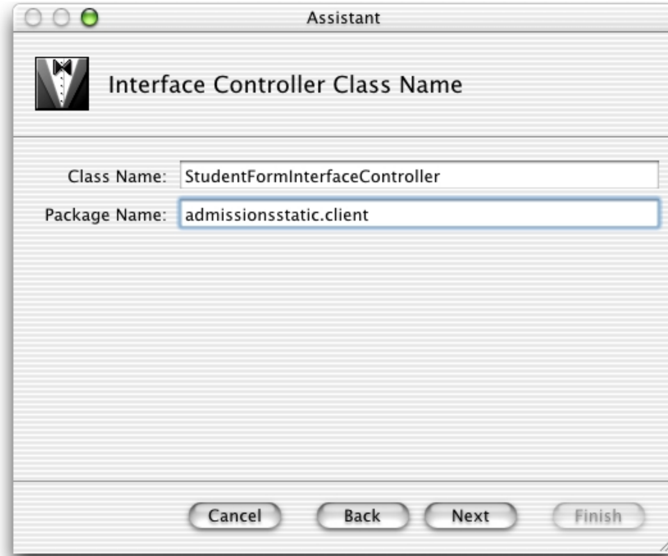
You create nondirect Java Client applications in Project Builder using the Java Client Application project type.

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 6-1. Make sure the package name is `admissions.client`. When creating Java Cilent interfaces, you must always specify the correct package name.

© Apple Computer, Inc. May 2002

Figure 6-1 Name the interface controller



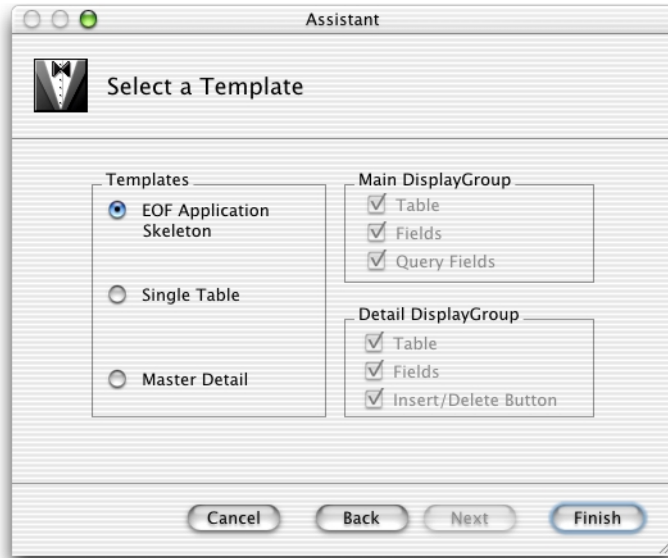
Add the `Admissions.eomodeld` file when prompted.

Choose the fourth option in the Choose Download Classes dialog (Download main bundle and custom framework classes).

In the Select a Template pane, select EOF Application Skeleton as shown in Figure 6-2.

© Apple Computer, Inc. May 2002

Figure 6-2 Choose a template for the interface controller

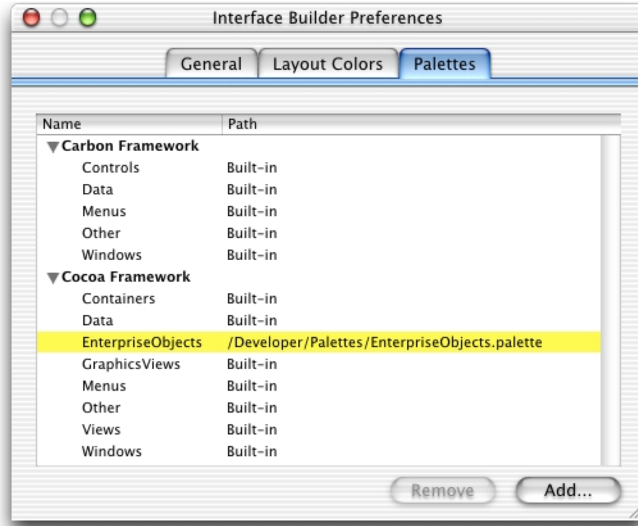


For Java Client applications, Project Builder creates an Interface Builder file (nib) and its associated Java class. By default, it's grouped in the Interfaces group. Double-click `StudentFormInterfaceController.nib` to open the file in Interface Builder.

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 Interface Builder's preferences window as shown in Figure 6-3 (page 170).

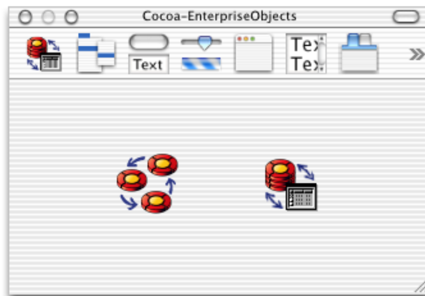
© Apple Computer, Inc. May 2002

Figure 6-3 Interface Builder palettes



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 6-4.

Figure 6-4 Enterprise Objects palette



© Apple Computer, Inc. May 2002

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.

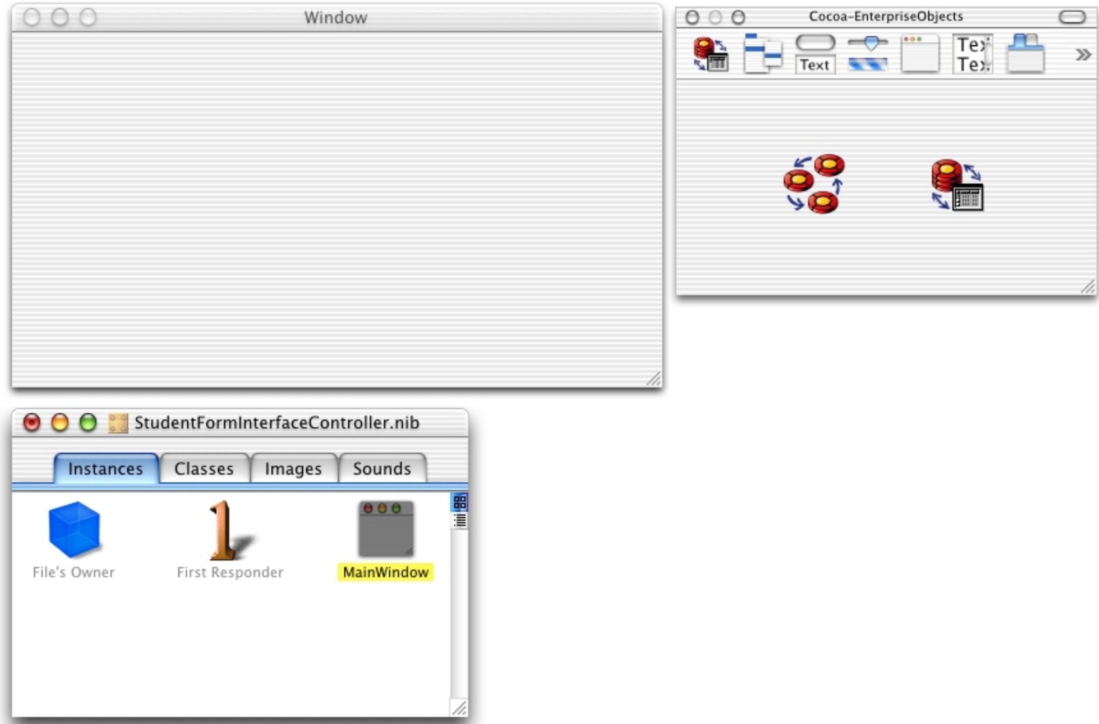
The associations and connections you can make in Interface Builder make it the best tool for developing completely custom user interfaces for Java Client applications. You can write completely custom Java Client user interfaces in raw Swing or by using other third-party tools, but then you'll have to 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 simple 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 6-5 (page 172).

© Apple Computer, Inc. May 2002

Figure 6-5 The Interface Builder environment



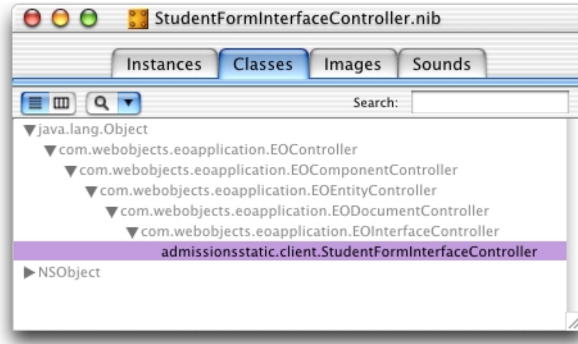
Prepare the Nib File

Before adding to the nib file, you may need to associate it with its controller class. This should happen for you but if it's not, you must make the association manually.

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 6-6.

© Apple Computer, Inc. May 2002

Figure 6-6 Classes pane in the nib file window

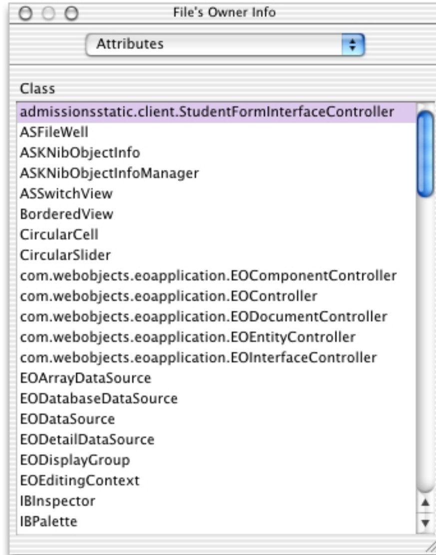


Click `com.webobjects.eoapplication.EOInterfaceController` in the classes list and press Return. This subclasses EOInterfaceController and thus the new class inherits its targets and outlets. The name of the new subclass is the fully qualified name of the nib file, `admissions.client.StudentFormController`, as shown in Figure 6-6.

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 `admissions.client.StudentFormController` as shown in Figure 6-7.

© Apple Computer, Inc. May 2002

Figure 6-7 Assign the custom subclass to File's Owner

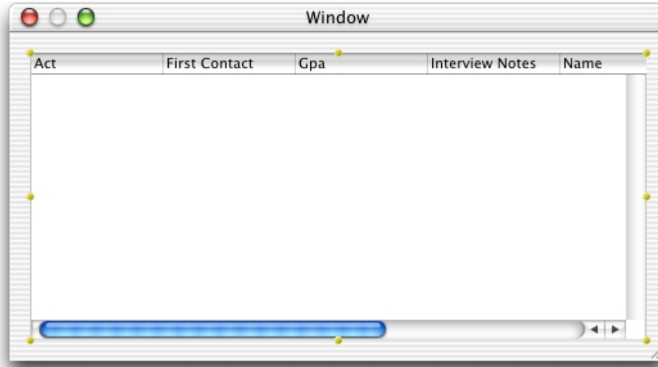


Integrate the Model

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 6-8 (page 175).

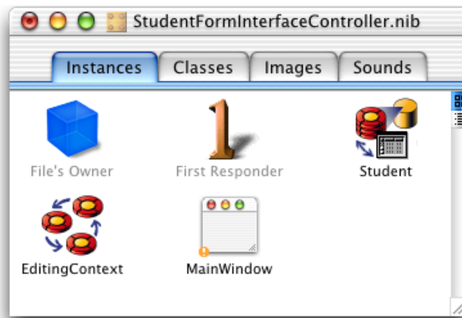
© Apple Computer, Inc. May 2002

Figure 6-8 The Student entity dragged into Interface Builder



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 to the nib file window. The nib file window should appear as in Figure 6-9.

Figure 6-9 Display group and editing context



© Apple Computer, Inc. May 2002

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 6-10. This option is important because it allows data to be fetched from the database when the application starts up.

Figure 6-10 Display group options in Interface Builder

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 “Task: Using Pop-up Menus In Nib Files” (page 275).

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.

© Apple Computer, Inc. May 2002

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.

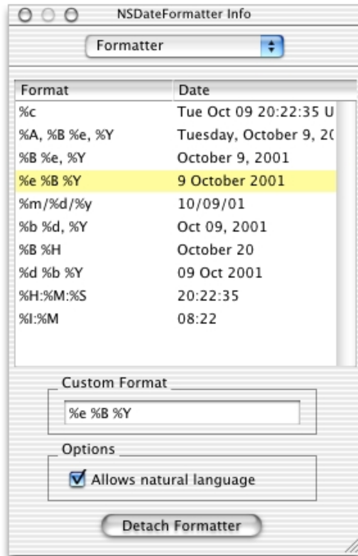
To change the formatters, double-click one of the columns and bring up the Info window. Choose Formatters from the pop-up menu and select a formatter with a decimal point for the `Gpa` column as shown in Figure 6-11.

Figure 6-11 Choose a formatter for the `Gpa` column

For the `FirstContact` column, select a simple date format as shown in Figure 6-12 (page 178).

© Apple Computer, Inc. May 2002

Figure 6-12 Choose a formatter for the `FirstContact` column

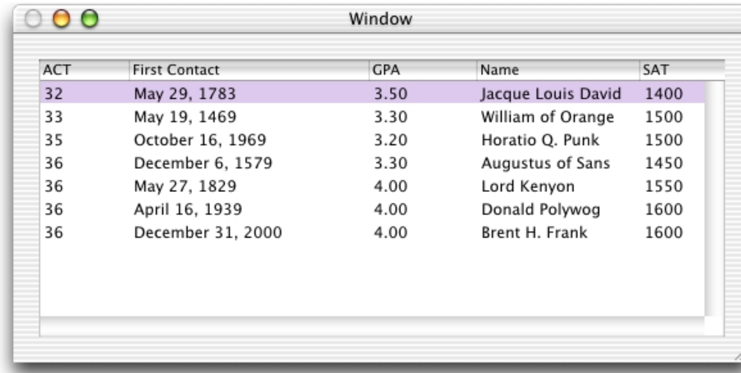


Finally, capitalize the column names so that they're as shown in Figure 6-13.

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.

© Apple Computer, Inc. May 2002

Figure 6-13 Testing the application



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

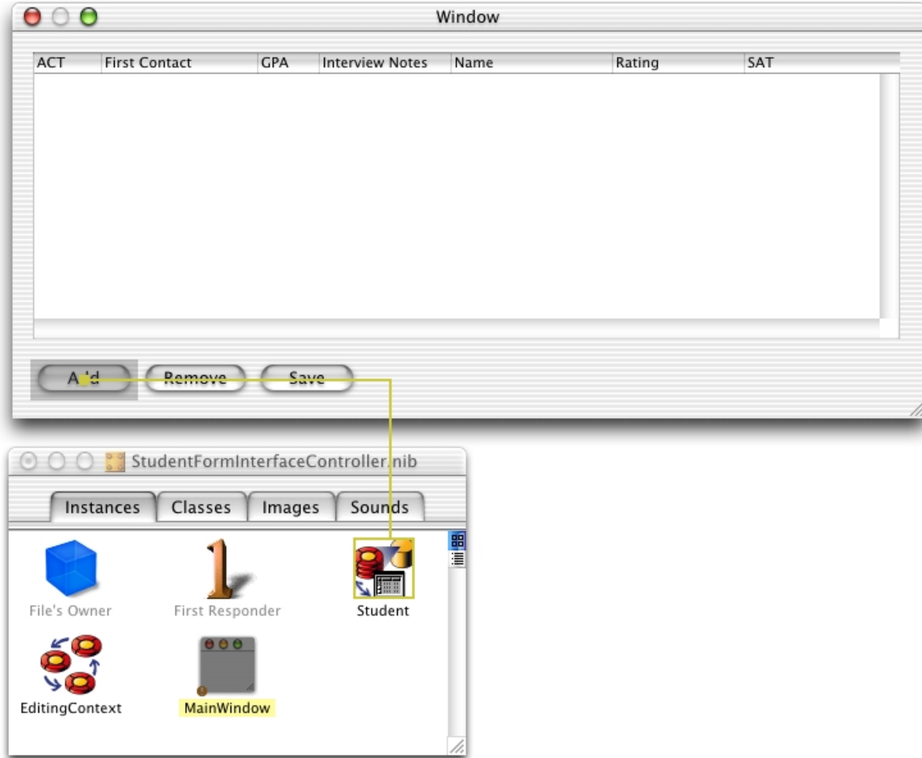
You can add basic behavior to your application, such as adding, deleting, and saving objects, without writing a line of code. This is possible because the EODisplayGroup, EOEditingContext, and EOInterfaceController objects in Interface Builder have predefined action methods that you can use to trigger 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 Cocoa-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 6-14 (page 180) and Figure 6-15 (page 181).

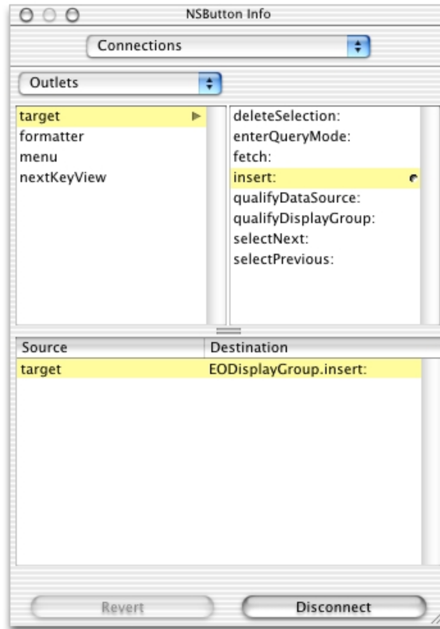
© Apple Computer, Inc. May 2002

Figure 6-14 Connect the Add button to the `insert` method of the Student EODisplayGroup



© Apple Computer, Inc. May 2002

Figure 6-15 Select the `insert` method



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

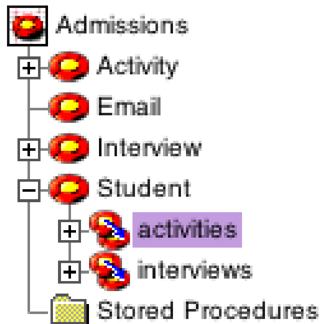
To express the relationships in your EOModel, you use a master-detail interface. This interface includes a master table that holds records for the source of the relationship and a detail table that holds records for the destination. As individual records in the master table are selected, the contents of the detail table change to show the records that correspond to the selection in the master table.

© Apple Computer, Inc. May 2002

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 6-16 The `activities` relationship in the Student entity

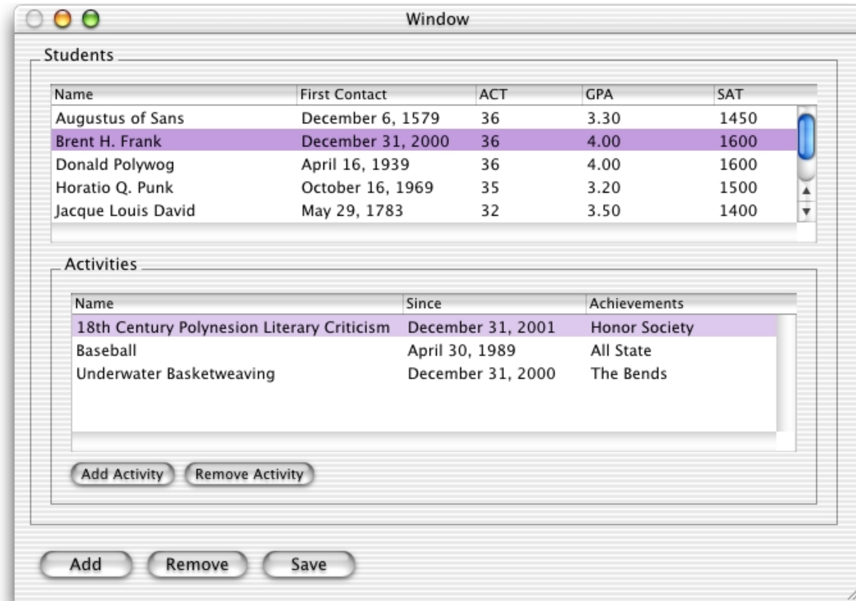


Reconnect the Add and Remove buttons to the Student EODisplayGroup. Add two buttons for detail part of the relationship to add and remove activities. Connect them to the activities display group. Add the formatters for the columns as you did earlier.

Test the master-detail interface by choosing Test Interface from the File menu. Figure 6-17 shows the master-detail interface.

© Apple Computer, Inc. May 2002

Figure 6-17 A master-detail interface

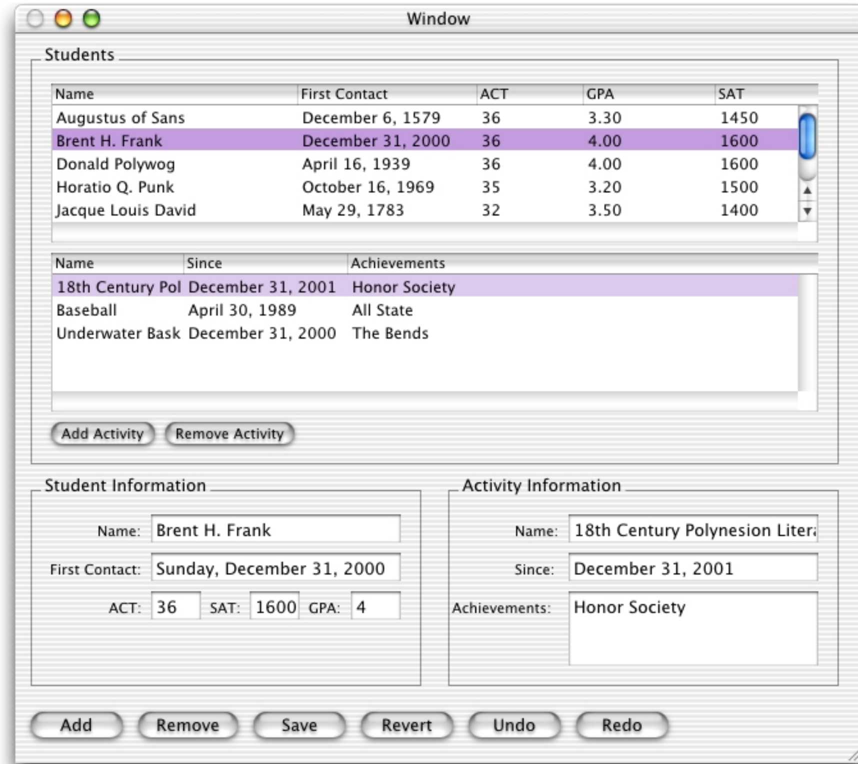


The master-detail interface you just created can be improved. Although you can add new records by entering text directly in the table columns, it would be nice to provide text fields for doing the same thing. Also, you should take advantage of more built-in features of the technology, such as reversion, undo, and redo.

It's easy to add widgets in Interface Builder. Simply drag widgets from the Interface Builder palette onto the window. Figure 6-18 illustrates the complete widget set of text fields, text areas, and buttons for the master-detail interface.

© Apple Computer, Inc. May 2002

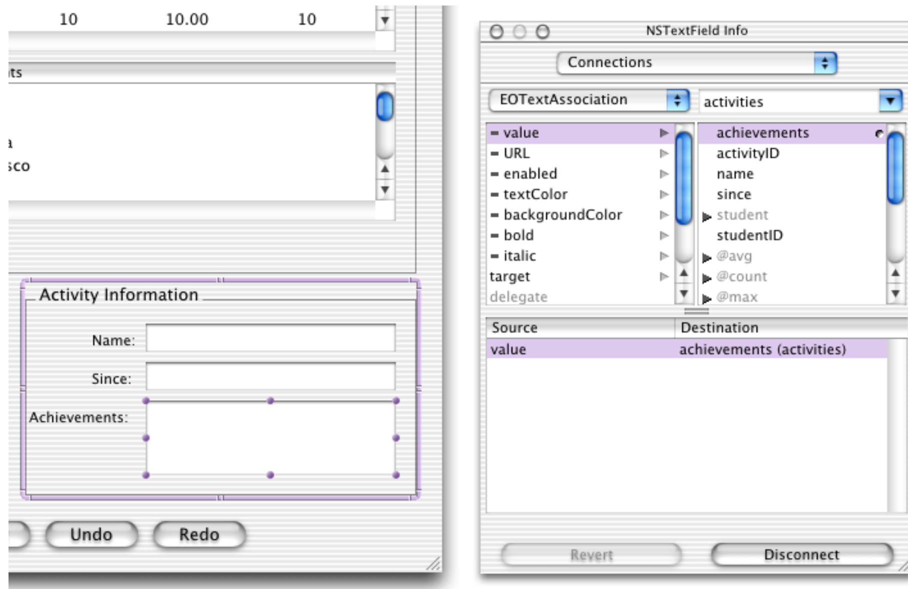
Figure 6-18 Complete widget set for the master-detail interface



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 6-19. 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.

© Apple Computer, Inc. May 2002

Figure 6-19 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.

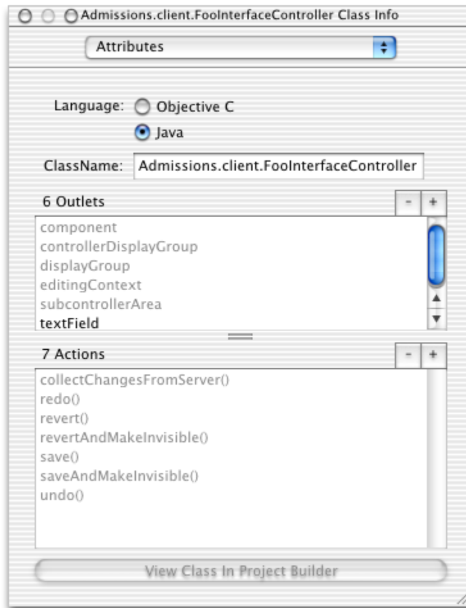
Programmatic Access to Interface Components

It's common to want programmatic access to user interface components in nib files. In the Cocoa world, 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.

© Apple Computer, Inc. May 2002

In a nib file, select File's Owner in the nib file window, switch to the Classes pane, and select 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 6-20.

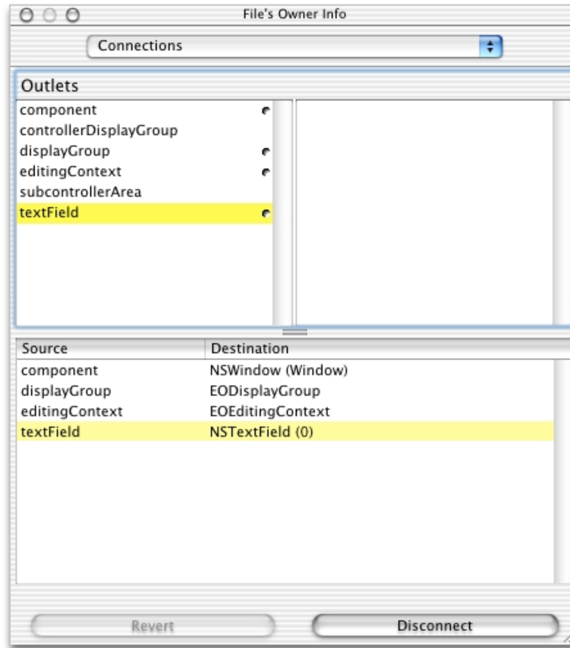
Figure 6-20 Add an outlet



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 6-21.

© Apple Computer, Inc. May 2002

Figure 6-21 Connect the new outlet



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.

© Apple Computer, Inc. May 2002

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.

© Apple Computer, Inc. May 2002

CHAPTER 7

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 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) do 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:

© Apple Computer, Inc. May 2002

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 "Task: Localizing Dynamic Components" (page 255)) 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.

© Apple Computer, Inc. May 2002

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 deal with 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 will confuse 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.

© Apple Computer, Inc. May 2002

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`, `modalDialog`, and potentially 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

© Apple Computer, Inc. May 2002

- `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. The two categories of controllers are:

- `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 to 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).

© Apple Computer, Inc. May 2002

Open EOWindow.wo in WebObjects Builder. (You can find it in /System/Library/Frameworks/JavaEOGeneration.framework/Resources).

EOWindow.wo contains an .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.

© Apple Computer, Inc. May 2002

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";
}
```


© Apple Computer, Inc. May 2002

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

`componentName` and `componentNameKey` 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.

© Apple Computer, Inc. May 2002

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.

© Apple Computer, Inc. May 2002

© Apple Computer, Inc. May 2002

CHAPTER 8

Task: 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 which 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 "Distribution Layer" (page 107) 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`

© Apple Computer, Inc. May 2002

See “New D2WComponent” (page 216) 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 285) to learn how to load a nib file when an application starts up. See “Task: Adding Custom Menu Items” (page 215) 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`

© Apple Computer, Inc. May 2002

This rule points the default specifications to a D2WComponent called "BlankSpecifications." The HTML file of the BlankSpecifications component is simply an empty array:

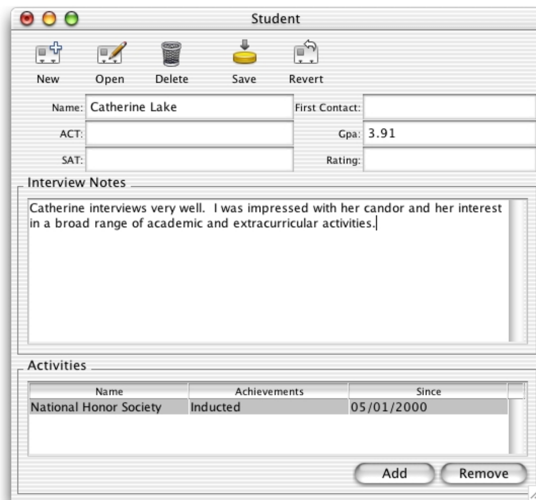
```
<ARRAY></ARRAY>
```

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 "Task: Building a Login Window" (page 285) 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 8-1. These actions are: insert, open, delete, save, and revert.You want to disable the buttons that invoke some of these actions.

Figure 8-1 Default actions in a form window



© Apple Computer, Inc. May 2002

Solution: Use the rule system to override the default behavior.

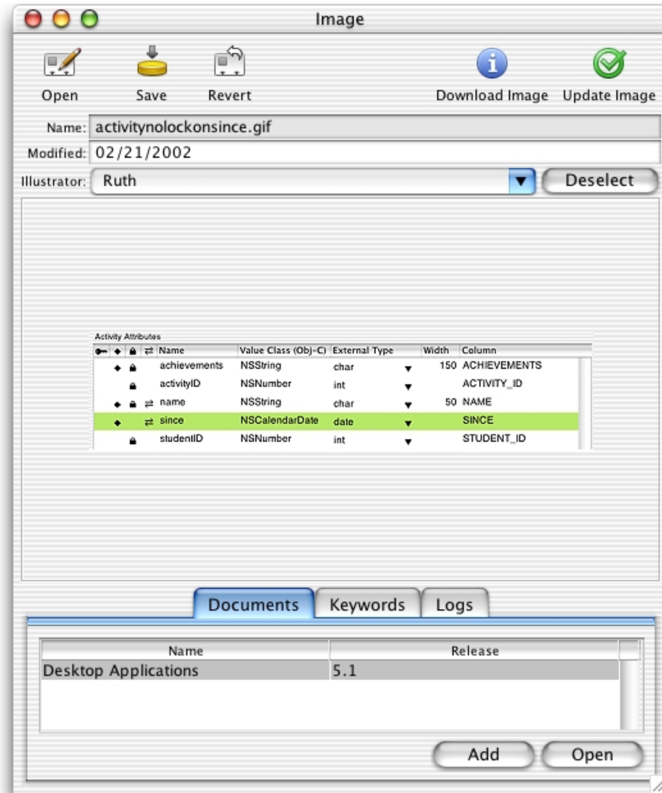
The rule system provides a key to disable certain actions. By providing the names of the actions you wish to disable as the right-hand side value of this rule, those actions are disabled in all dynamically generated controllers. This and many other rules have no effect on frozen XML or frozen interface 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 8-2.

© Apple Computer, Inc. May 2002

Figure 8-2 Disabled actions in a form window



To understand how the additional buttons Download Image and Update Image are added, refer to "Task: Adding Custom Actions to Controllers" (page 209).

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 237) to learn how to write and use custom controller classes.

© Apple Computer, Inc. May 2002

© Apple Computer, Inc. May 2002

Task: Using the Controller Factory Programmatically

Much of the magic behind Direct to Java Client applications happens in the controller factory, the class `com.webobjects.eogeneration.client.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 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 a good hour 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.

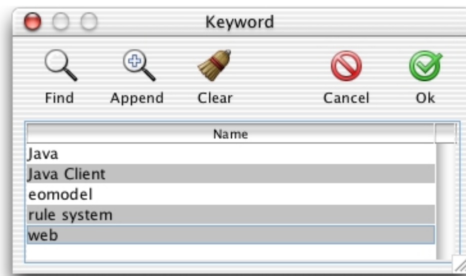
In a client-side view class (such as the CustomFormController class in "Extend a Controller Class" (page 158) or a subclass of another core controller class), add the import statement for `com.webobjects.eogeneration.client`. This package contains the controller factory. Then, in the action method that triggers the selection, add this invocation on the controller factory:

© Apple Computer, Inc. May 2002

```
EOControllerFactory.sharedControllerFactory().selectWithEntityName("<entity
name>", 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 9-1.

Figure 9-1 Select dialog



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.

© Apple Computer, Inc. May 2002

Task: Using the Controller Factory Programmatically

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.client`. 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("imageQ
uery");
```

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.

Were you to implement this feature without using the controller factory or the rule system, you could spend a good hour 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.

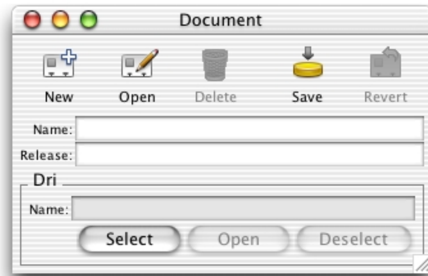
In a client-side view class (not a model class or a controller class), add the import statement for `com.webobjects.eogeneration.client`. This package contains the controller factory. Then, in the action method that triggers the selection, add this invocation on the controller factory:

© Apple Computer, Inc. May 2002

```
EOControllerFactory.sharedControllerFactory().insertWithEntityName("Documen
t");
```

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 9-2.

Figure 9-2 Form window from controller factory



© Apple Computer, Inc. May 2002

Task: 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 topic describes all the possibilities.

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 158) in the chapter "Advanced Tutorial" (page 119).

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

© Apple Computer, Inc. May 2002

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 10-1 (page 210).

Listing 10-1 Subclassing EOFormController

```
package assetmanager.client;

import javax.swing.*;
import com.webobjects.foundation.*;
import com.webobjects.eocontrol.*;
import com.webobjects.eoapplication.*;
import com.webobjects.eogeneration.client.*;
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));
```


© Apple Computer, Inc. May 2002

```
    }

    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.

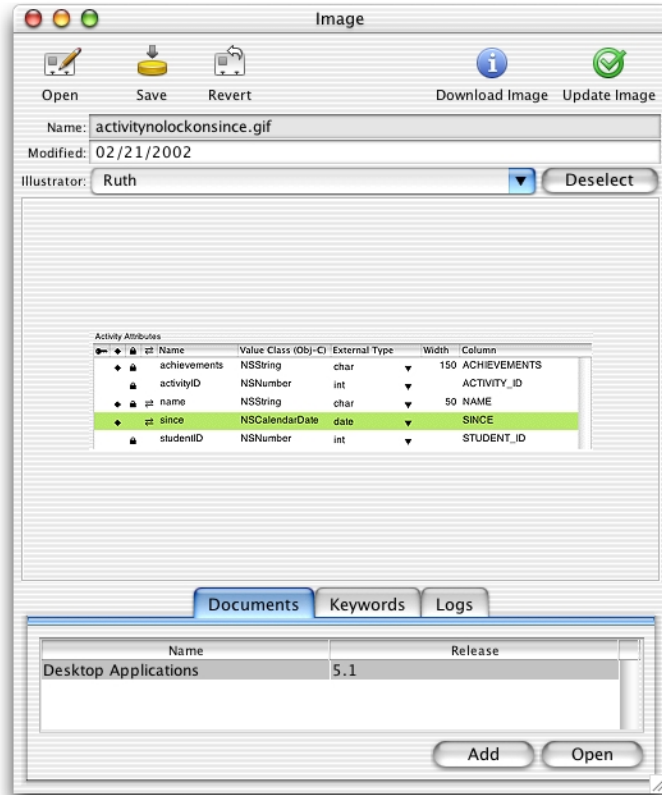
To use this class in all form windows throughout the application, you need only write a simple rule:

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 10-1.

© Apple Computer, Inc. May 2002

Figure 10-1 Image form window with new actions



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 "Task: Restricting Access to an Application" (page 199).

© Apple Computer, Inc. May 2002

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 209), except that you subclass EOController. Listing 10-2 shows the code for the class that adds an action that displays a simple information dialog.

Listing 10-2 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.client.*;

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());
    }
```


© Apple Computer, Inc. May 2002

```
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`

© Apple Computer, Inc. May 2002

Task: Adding Custom Menu Items

There are many ways to add custom menu items to a Direct to Java Client application. This topic describes the most common mechanisms.

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 Appendix , "EOActions XML Descriptions", 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. These actions 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. These actions 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.

© Apple Computer, Inc. May 2002

`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 "Task: Restricting Access to an Application" (page 199) 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 topic for adding one to your 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

© Apple Computer, Inc. May 2002

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 11-1.

Listing 11-1 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.

© Apple Computer, Inc. May 2002

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 216)), the XML description for an `APPLICATIONACTION` 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 and that 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 312).

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:

© Apple Computer, Inc. May 2002

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 "Advanced Tutorial" (page 119), 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.

© Apple Computer, Inc. May 2002

© Apple Computer, Inc. May 2002

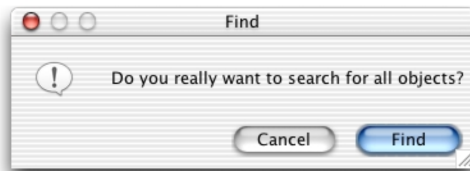
Task: Customizing With Common Rules

This chapter provides examples of some common rules you can use to customize applications.

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 12-1. This behavior is intended to warn users about performing unqualified queries of the data store, which could fetch hundreds or thousands of records.

Figure 12-1 Confirm dialog on unqualified queries



Solution: Use the rule system to override the default behavior.

© Apple Computer, Inc. May 2002

The confirmation dialog results from user actions in query controllers. Whenever you want to modify the behavior of a controller in a Direct to Java Client application, you should first consult Appendix A, "XML Description of Classes and Actions". 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`

© Apple Computer, Inc. May 2002

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`

© Apple Computer, Inc. May 2002

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.client.*;
import com.webobjects.eointerface.swing.*;

public class CustomController extends EOWidgetController {

    public CustomController(EOXMLUnarchiver unarchiver) {
        super(unarchiver);
    }
```


© Apple Computer, Inc. May 2002

```
    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 "XML Description of Classes and Actions" (page 297). 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`

© Apple Computer, Inc. May 2002

© Apple Computer, Inc. May 2002

Task: Freezing XML User Interfaces

You can use the tutorial project you created earlier in Chapter 5, “Advanced Tutorial” (page 119), 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

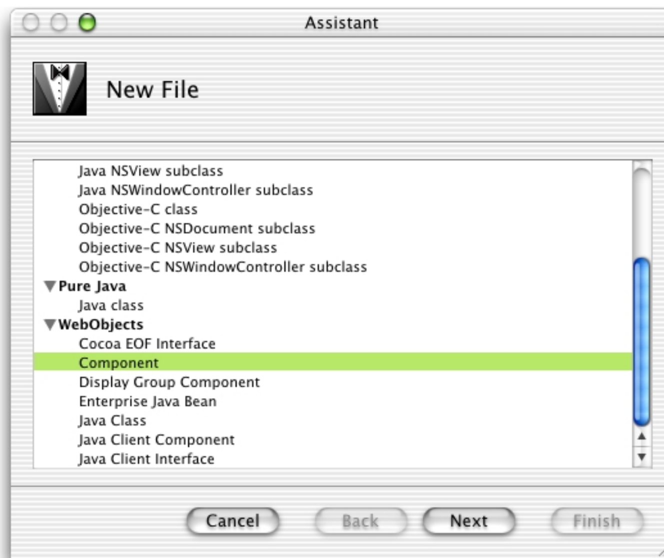
© Apple Computer, Inc. May 2002

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 13-1. Do not select Display Group Component or Java Client Component.

Figure 13-1 Select Component as the file type



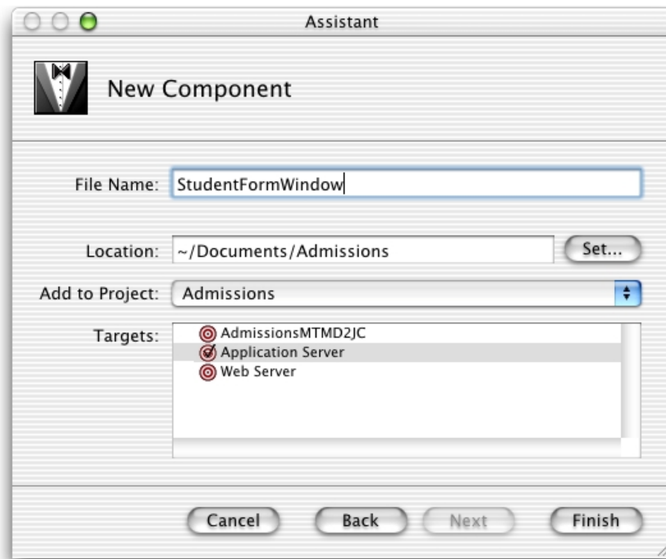
2. Name the new component “StudentFormWindow” and make sure Application Server is the selected target, as shown in Figure 13-2.

© Apple Computer, Inc. May 2002

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 13-2 Name new component “StudentFormWindow”



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 13-1.

© Apple Computer, Inc. May 2002

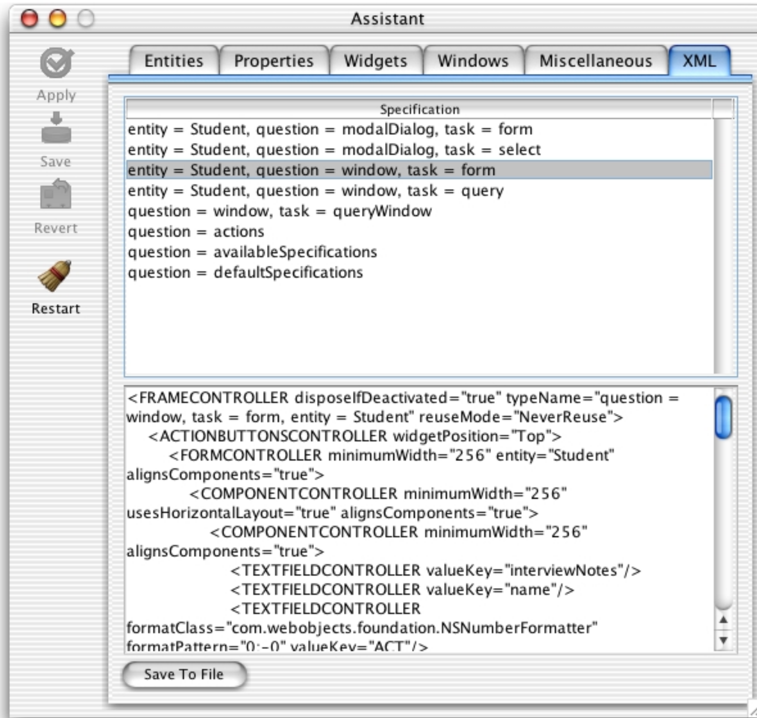
Listing 13-1 Change the superclass of StudentFormWindow to D2WComponent

```
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, select entity=Student, question=window, task=form. This puts the XML description for that selection in the XML window as shown in Figure 13-3.

© Apple Computer, Inc. May 2002

Figure 13-3 XML description of Student entity, form window



7. Copy the whole XML specification and paste it into the `StudentFormWindow.html` file in Project Builder. `StudentFormWindow.html` is in the StudentFormWindow component.

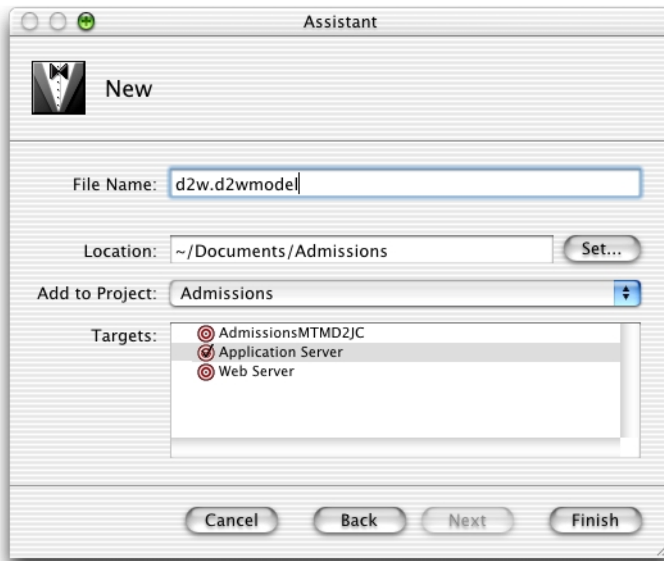
8. In Project Builder, select the Resources group and choose File > New File and select Empty File. Name the file "d2w.d2wmodel" and make sure Application Server is the selected target, as shown in Figure 13-4 (page 232). This file will hold custom rules you write. Skip this step if your project already has a `d2w.d2wmodel` file.

   You need to make a new `.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.

© Apple Computer, Inc. May 2002

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 13-4 Make a new rule file for custom rules



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.

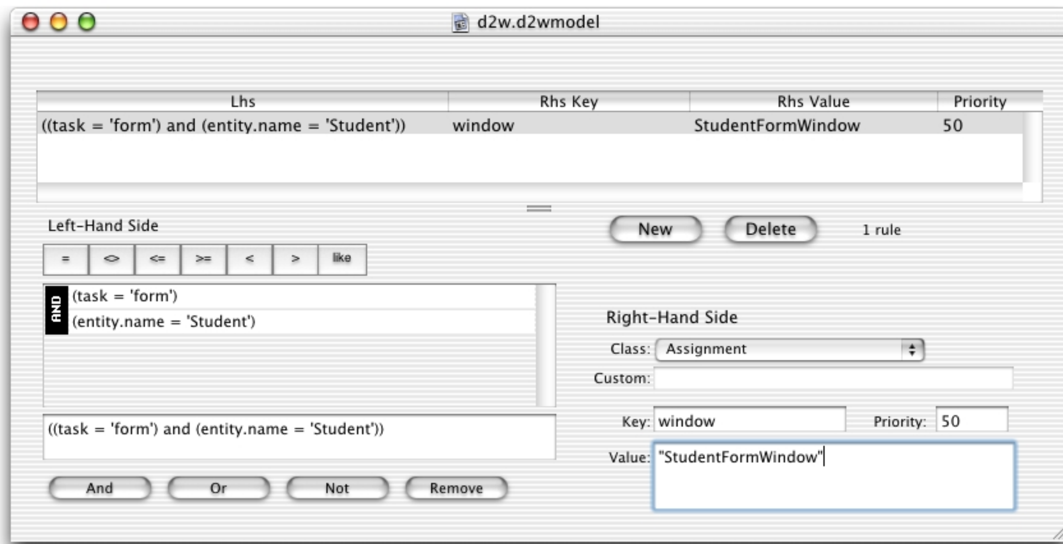
© Apple Computer, Inc. May 2002

In this case, if the query 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 mentioned earlier, 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 13-5 for clarity. For an explanation of rule system priorities, see "Rule System Priorities" (page 191).

Figure 13-5 Add a rule to use frozen XML



© Apple Computer, Inc. May 2002

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 13-2):

```
<BOXCONTROLLER usesTitleBorder="false" highlight="true"
border="RaisedBezel">
```

The beginning of the `StudentFormWindow.html` file should look like Listing 13-2. 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 13-2.

© Apple Computer, Inc. May 2002

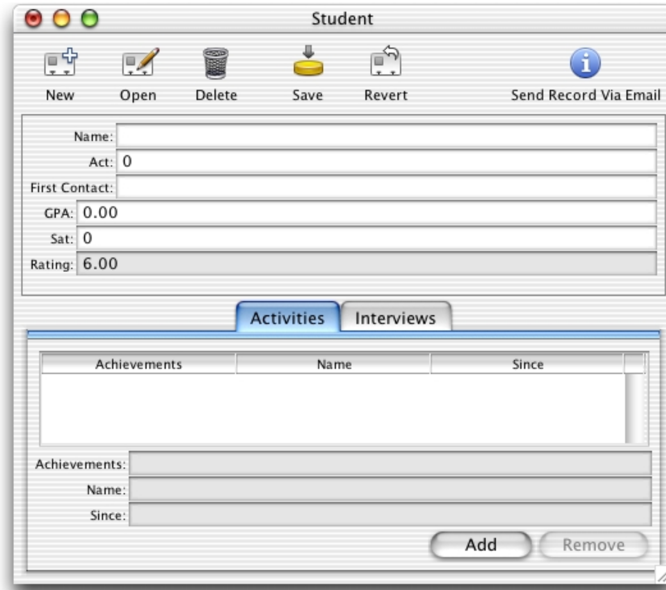
Listing 13-2 StudentFormWindow.html (frozen XML)

```
<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"
                   border="RaisedBezel">                                        //1
                <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>                                                //2
```

Figure 13-6 shows an example of a Student form window with the new `BOXCONTROLLER`.

© Apple Computer, Inc. May 2002

Figure 13-6 Student form window with BOXCONTROLLER tag



Note: You can also use most of the WebObjects dynamic elements in frozen XML components such as WOConditional, WORepetition, and WOString.

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.

© Apple Computer, Inc. May 2002

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.client.*;

public class NewController extends EOController {

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

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


© Apple Computer, Inc. May 2002

```
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 runInfoDialog() {
        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.

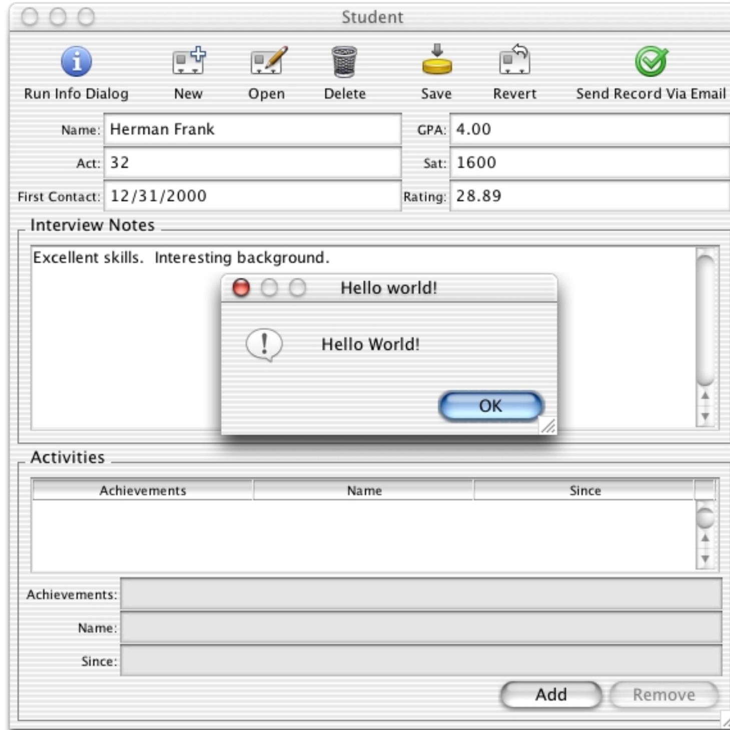
After writing the custom controller class, you must include it in a frozen XML component:

```
<CONTROLLER className="businesslogic.client.NewController">
```

The implementation of the action method in this example simply puts up a dialog as shown in Figure 13-7.

© Apple Computer, Inc. May 2002

Figure 13-7 Action in custom controller class



© Apple Computer, Inc. May 2002

© Apple Computer, Inc. May 2002

C H A P T E R 1 4

Task: Mixing Static and Dynamic User Interfaces

In this chapter, you'll learn how to integrate static interfaces made in Interface Builder within 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.

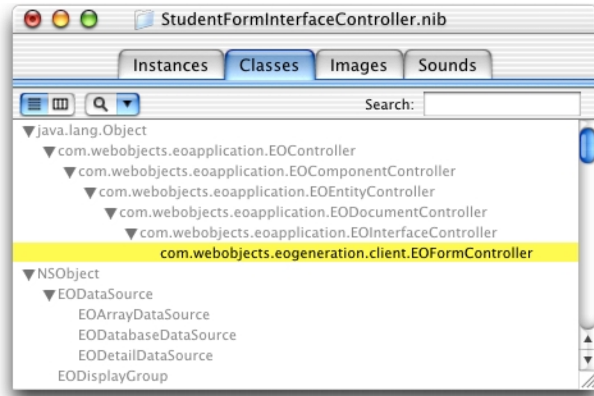
Preparing the Nib for Freezing

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 14-1.

© Apple Computer, Inc. May 2002

Figure 14-1 Classes pane in the nib file window



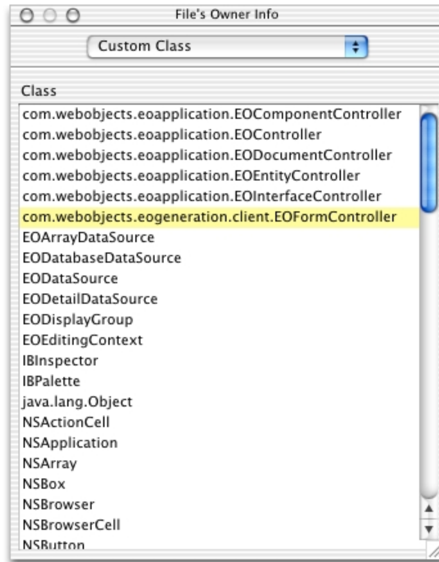
To use a nib file in a Direct to Java Client application, the class must be the exact class you use to load the interface. So, if you want to use the nib file in a form controller, you'll use EOFormController. To use it in a query controller, use EOQueryController. These classes are not automatically 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 inherits the targets and outlets you need for the new class. Name the new subclass `com.webobjects.eogeneration.client.EOFormController` as shown in Figure 14-1.

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.client.EOFormController` as shown in Figure 14-2.

© Apple Computer, Inc. May 2002

Figure 14-2 Assign the custom subclass to File's Owner



Finally, associate the nib file's controller class (it's associated `.java` class) with the same package with which other client-side classes in your application are associated:

```
package edu.admissions.client;
```

Save the nib file.

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.

© Apple Computer, Inc. May 2002

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 `StudentFormController`. 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.

© Apple Computer, Inc. May 2002

CHAPTER 15

Task: Using Custom Views in Interface 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.

Custom Views

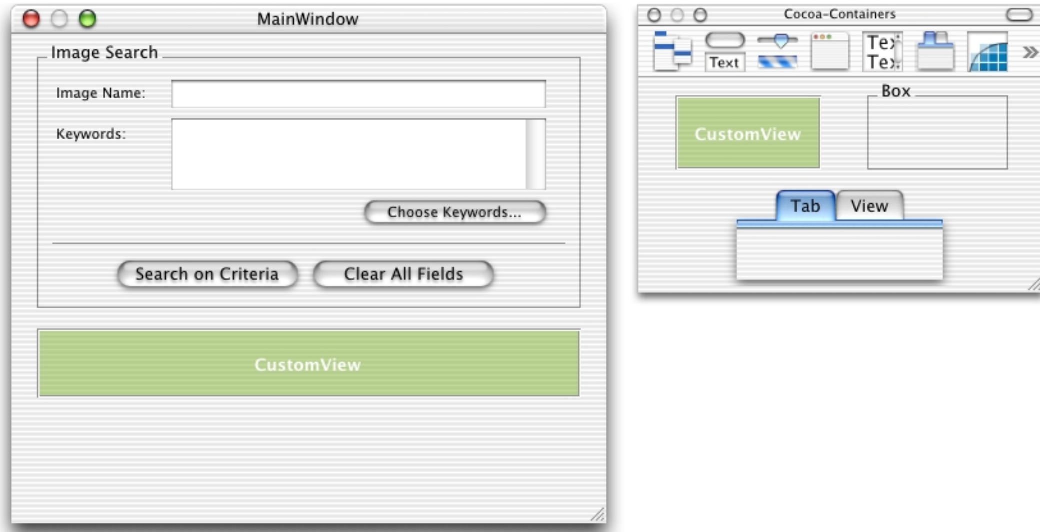
Problem: You want to add an unsupported view in an interface file such as `javax.swing.JProgressBar`.

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 Cocoa-Containers palette. Figure 15-1shows this palette and a custom view placed in the main window.

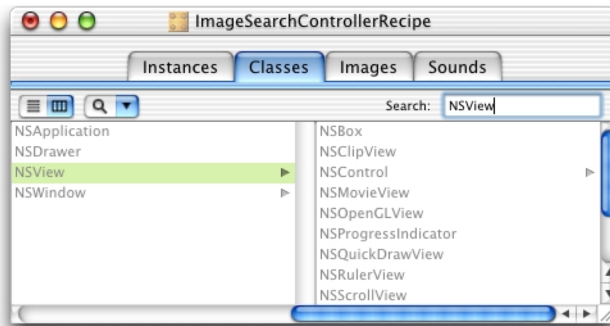
© Apple Computer, Inc. May 2002

Figure 15-1 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 15-2.

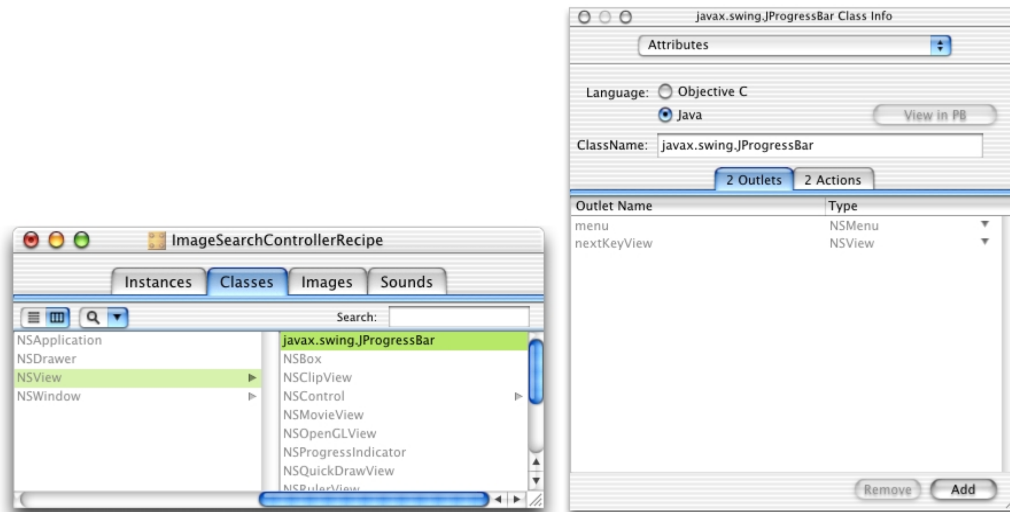
Figure 15-2 Find NSView in class hierarchy



© Apple Computer, Inc. May 2002

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 15-3. If the view represents a custom Swing subclass, specify the fully-qualified name of that subclass.

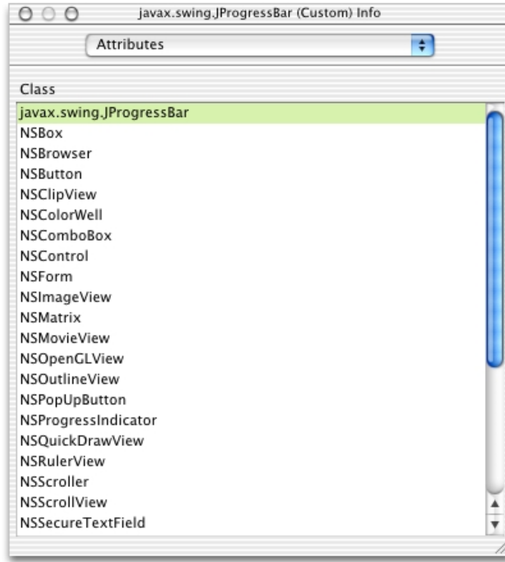
Figure 15-3 Name the custom view class



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 15-4.

© Apple Computer, Inc. May 2002

Figure 15-4 Associate custom view with NSView subclass



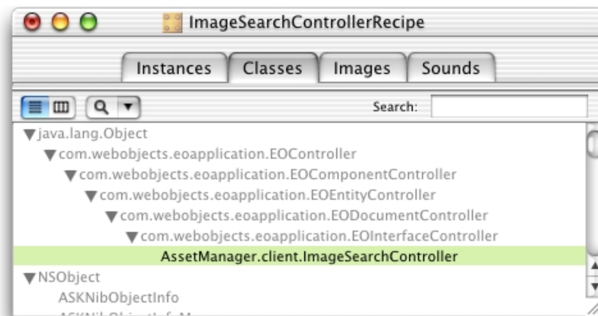
The name in the custom view should then change to the name of the new class, as shown in Figure 15-5.

© Apple Computer, Inc. May 2002

Figure 15-5 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 15-6.

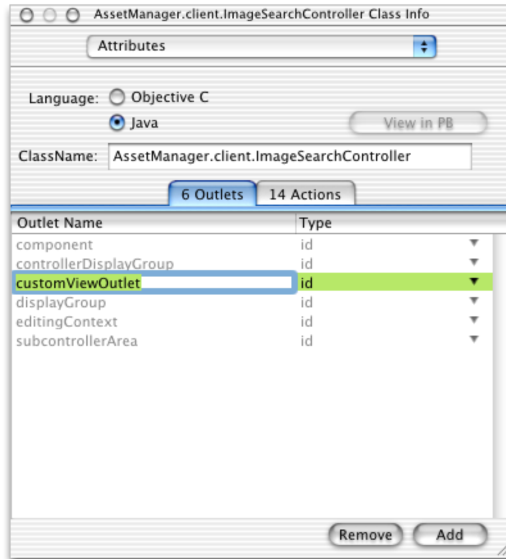
Figure 15-6 File's Owner's class



© Apple Computer, Inc. May 2002

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 15-7.

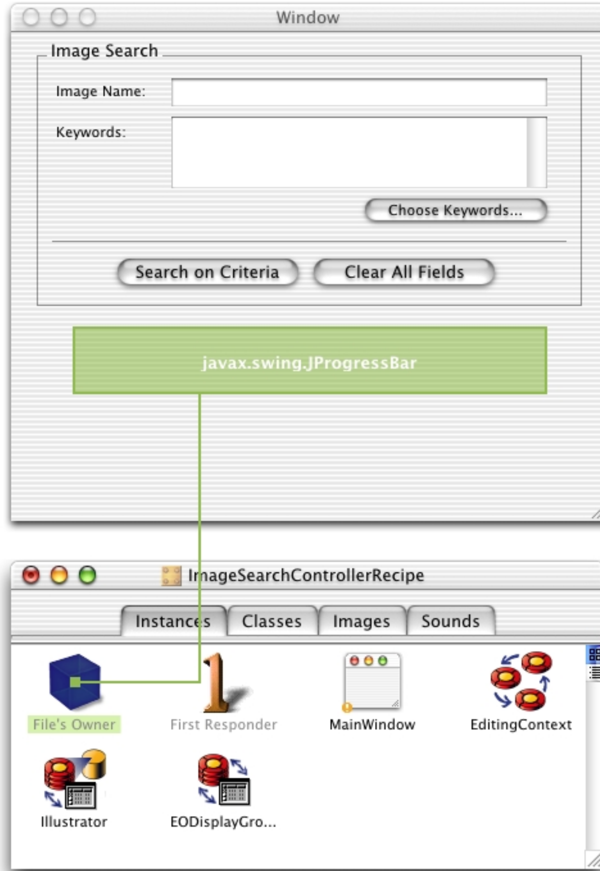
Figure 15-7 Add outlet to interface file



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 15-8.

© Apple Computer, Inc. May 2002

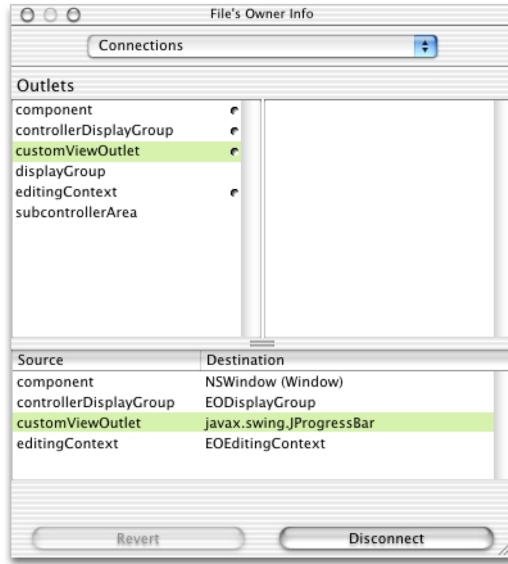
Figure 15-8 Connect new outlet to custom view



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 15-9.

© Apple Computer, Inc. May 2002

Figure 15-9 File's Owner's attributes



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 271) or check to see if the component is initialized by invoking `isComponentPrepared`.

© Apple Computer, Inc. May 2002

EOImageView

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 java.awt.event.*;
import com.webobjects.foundation.*;
import com.webobjects.eointerface.swing.*;
import com.webobjects.eogeneration.client.*;

public class CustomImageViewController extends EOImageView {

    public CustomImageViewController() {
        super();
        this.addMouseListener(new OpenRecord());
    }

    class OpenRecord extends MouseAdaptor {

        public void mouseClicked(MouseEvent e) {
            NSLog.out.appendln("image clicked");
        }

    }

}
```


© Apple Computer, Inc. May 2002

To use this custom class in an interface file, you subclass NSView as described in "Custom Views" (page 245) and name the subclass "com.mycompany.myapp.CustomImageViewController."

© Apple Computer, Inc. May 2002

Task: 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.

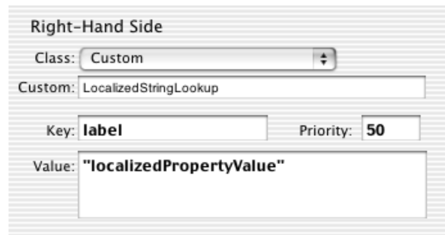
Most of the rules you write and use in the rule system have a right-hand side class of type Assignment as shown in Figure 16-1.

Figure 16-1 Right-hand side class of type Assignment

© Apple Computer, Inc. May 2002

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 16-2, the key `label` is resolved to the result of the method named `localizedPropertyValue` in the class LocalizedStringLookup.

Figure 16-2 Right-hand class of type Custom



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 this code into it:

Listing 16-1 LocalizedStringLookup class

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

public class LocalizedStringLookup extends DefaultAssignment {

    D2WContext d2wcontext;

    public LocalizedStringLookup(EOKeyValueUnarchiver unarchiver) {
        super(unarchiver);
    }
    public LocalizedStringLookup(String key, String value) { super(key,value); }
```


© Apple Computer, Inc. May 2002

```
    public static Object decodeWithKeyValueUnarchiver(EOKeyValueUnarchiver
          eokeyvalueunarchiver)  {
        return new LocalizedStringLookup(eokeyvalueunarchiver);
    }

    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("languages");

       String returnstr =
                WOApplication.application().resourceManager().stringForKey(displayName,
                "Localizable", displayName, null, languages);                      //2

        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

© Apple Computer, Inc. May 2002

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("languages")`), 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

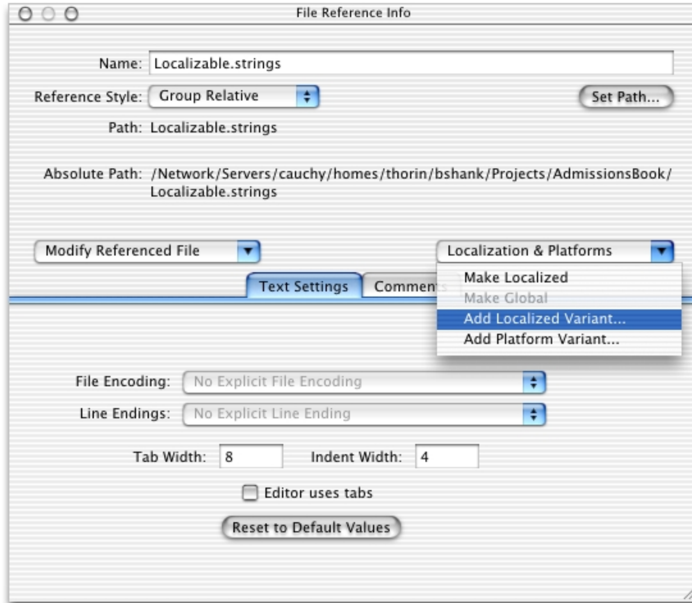
```
{
    "date" = "Fecha";
}
```

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 16-3.

© Apple Computer, Inc. May 2002

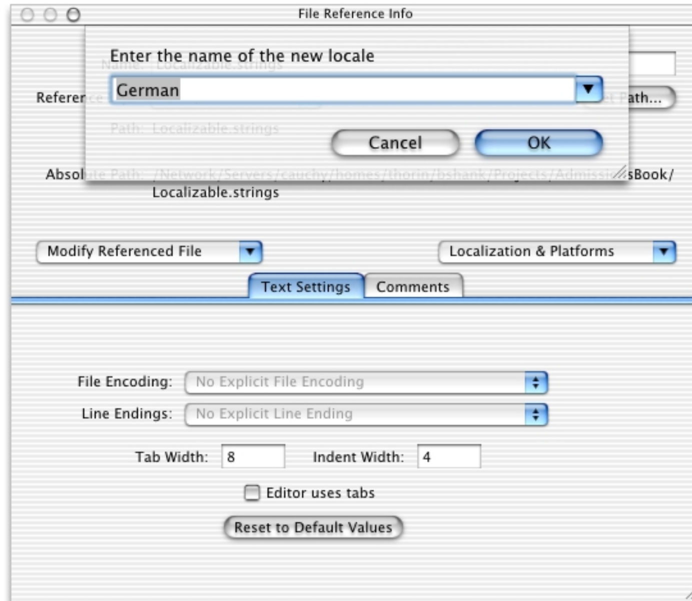
Figure 16-3 Add localized variant of `Localizable.strings` file



Add a localized variant for the language of your choice as shown in Figure 16-4. If the language is not listed, you can type it in the field underneath "Enter the name of the new locale."

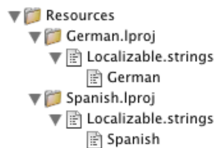
© Apple Computer, Inc. May 2002

Figure 16-4 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 16-5 shows German and Spanish localized variants in the Groups & Files list.

Figure 16-5 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 16-2.

© Apple Computer, Inc. May 2002

Listing 16-2 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. 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 16-2 (page 256).

© Apple Computer, Inc. May 2002

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 the same localization techniques described in "Localizing Property Labels" (page 255), adding string pairs for each string you want localized.

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";
```


© Apple Computer, Inc. May 2002

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 provided localization for into your `Localizable.strings` tables and localize them accordingly.

© Apple Computer, Inc. May 2002

© Apple Computer, Inc. May 2002

Task: Building Custom List Controllers

The public methods provided by the controller factory (`com.webobjects.eogeneration.client.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);
```


© Apple Computer, Inc. May 2002

```
    if (controller != null) {
        EOListController listController =
        (EOListController)f.controllerWithEntityName(controller,
          EOControllerFactory.List.class, entityName);
    listController.listObjectsWithFetchSpecification(fs);
    listController.setEditability(EOEditable.NeverEditable);
    listController.makeVisible();
    }
}
```


© Apple Computer, Inc. May 2002

Task: Using and Extending Image Views

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.

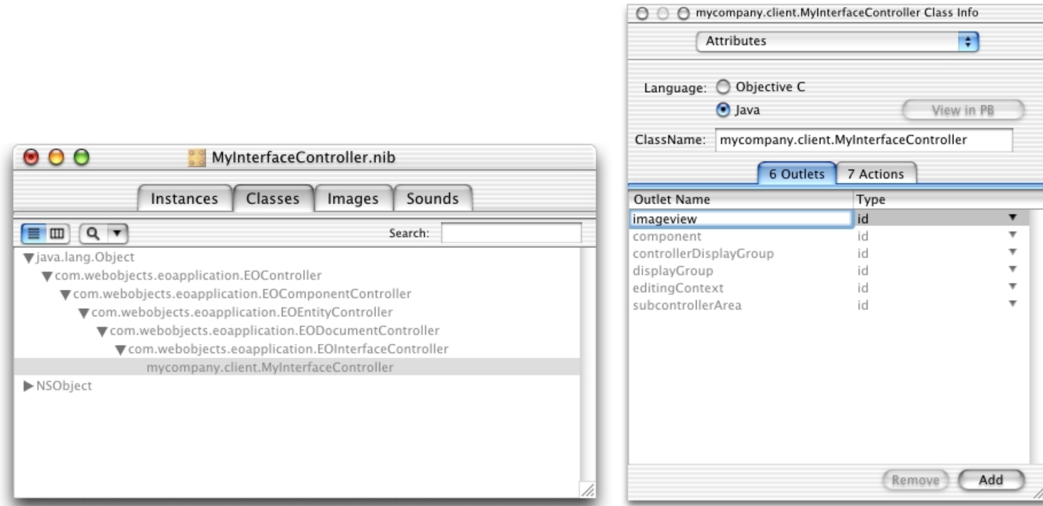
Adding Outlets

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 185). 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 18-1.

© Apple Computer, Inc. May 2002

Figure 18-1 Add a new outlet



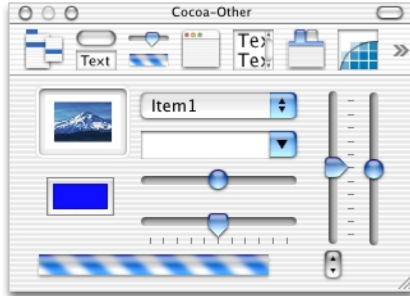
Now you're ready to add the image view widget to the interface.

Adding the Widget

You now need to add a widget to represent the image you want to display. The Cocoa-Other palette includes an image view widget that works for these purposes. If the widget palette isn't visible, choose Tools >Palettes > Show Palettes. Find the Cocoa-Other palette by clicking different buttons in the palettes toolbar. It is shown in Figure 18-2.

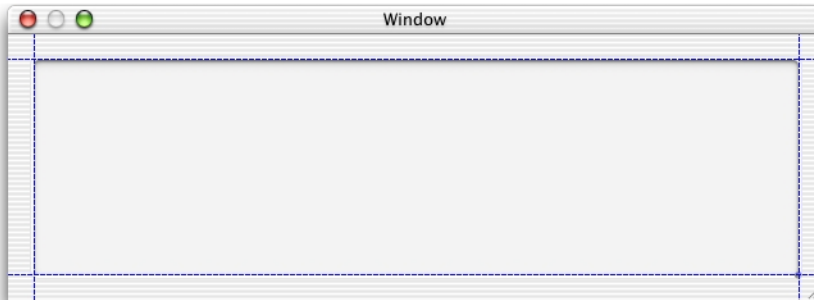
© Apple Computer, Inc. May 2002

Figure 18-2 Cocoa-Other palette



Drag the image view widget (the one in the upper-left corner of the Cocoa-Other 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 18-3.

Figure 18-3 Place widget with guides



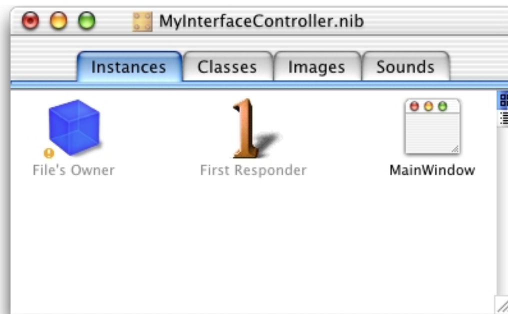
Now you need to connect the widget to the outlet you added to File's Owner.

© Apple Computer, Inc. May 2002

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 18-4.

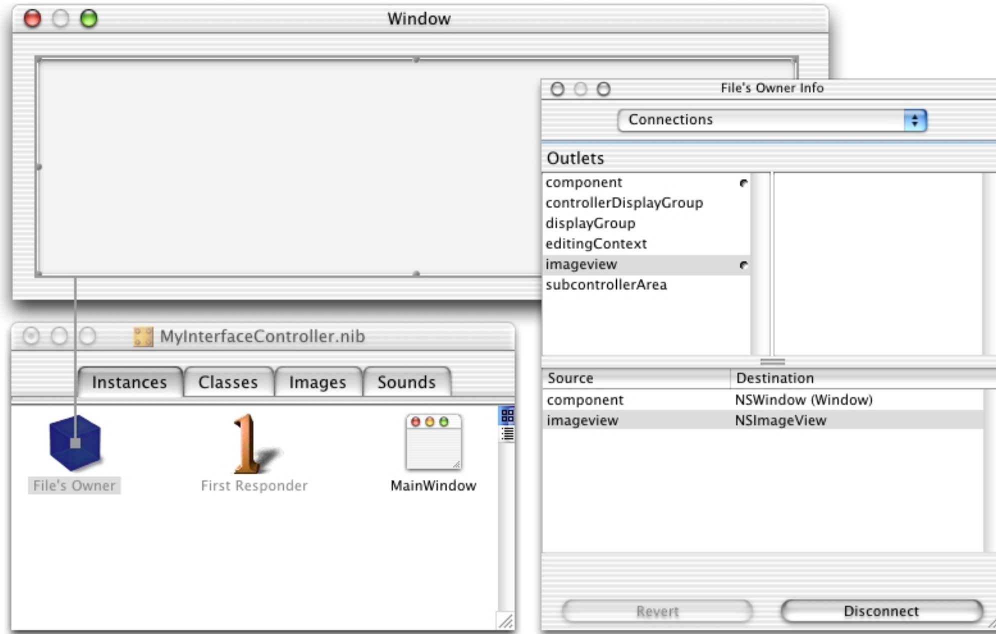
Figure 18-4 File's Owner icon with exclamation point



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 18-5.

© Apple Computer, Inc. May 2002

Figure 18-5 Connect outlet to widget



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 18-5.

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.

© Apple Computer, Inc. May 2002

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'll use the EOInterface Swing package (`com.webobjects.eointerface.swing.*`). So, add the import statement:

```
import com.webobjects.eointerface.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 18-1 (page 273).

© Apple Computer, Inc. May 2002

Listing 18-1 Overriding `controllerDidLoadArchive`

```
protected void controllerDidLoadArchive(NSDictionary namedObjects) {

    ImageIcon iIcon =
      (ImageIcon)EOUserInterfaceParameters.localizedIcon("iMac");    //1

    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 5.1, `EOUserInterfaceParameters.localizedIcon` retrieves images with extensions `gif`, `jpeg`, 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 codeline 1.

If successful, your interface file should load and display the specified image as shown in Figure 18-6.

Figure 18-6 Image in image view

© Apple Computer, Inc. May 2002

© Apple Computer, Inc. May 2002

Task: Using Pop-up Menus In Nib Files

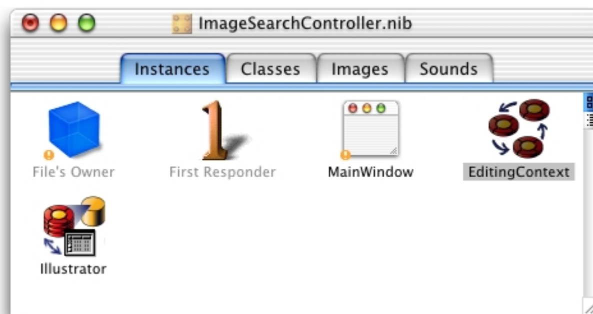
It's common to want to display pop-up menus in interface files that display a short list of enumeration values. This chapter describes how to connect a pop-up menu widget (`javax.swing.JComboBox`) to a display group and how to get the value of the selected object in the interface file's controller class.

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 19-1 (page 275) shows an entity called "Illustrator" as a display group in a nib file.

Figure 19-1 Illustrator entity in nib file



© Apple Computer, Inc. May 2002

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.

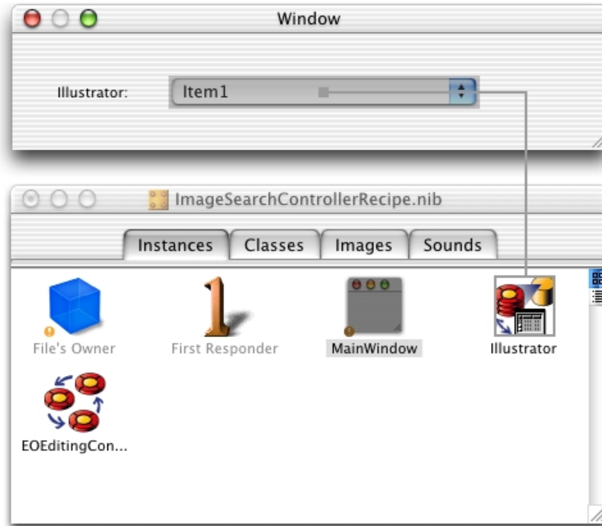
Now add a widget for the pop-up menu. You can find it in the Cocoa-Other palette, as shown in Figure 19-2. It's the widget that includes the text "Item1".

Figure 19-2 Cocoa-Other palette

Then, Control-drag from the widget to the display group for the entity containing the enumeration values, as shown in Figure 19-3.

© Apple Computer, Inc. May 2002

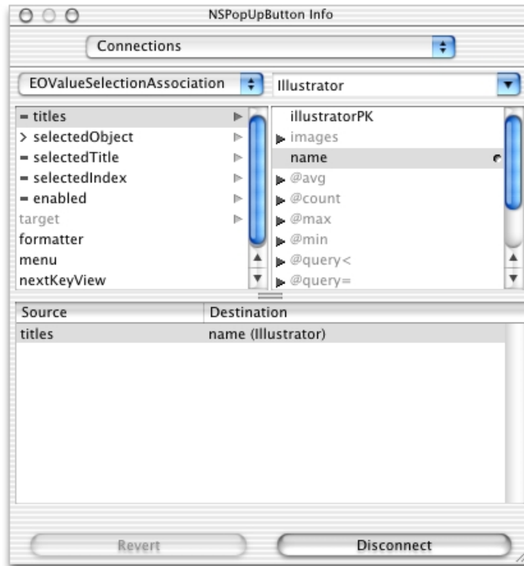
Figure 19-3 Connect widget to display group



This action displays the Info window so you can set the binding for the `titles` aspect of the EOValueSelectionAssociation. As shown in Figure 19-4, bind the `titles` aspect to the attribute of the entity that represents the enumeration value, `name` in the example shown here.

© Apple Computer, Inc. May 2002

Figure 19-4 Bind the title aspect to the appropriate attribute

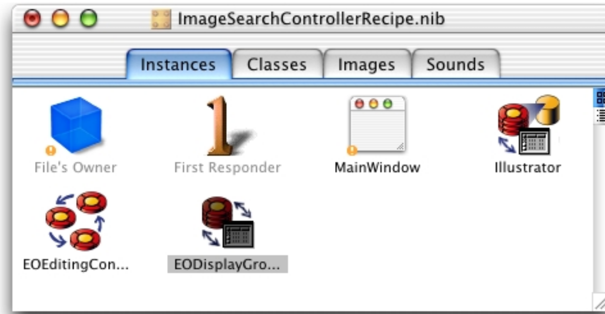


Save the nib and choose Test Interface from the File menu. You should see the values of the attribute bound to the `titles` aspect of the pop-up menu as items in that menu.

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 19-5.

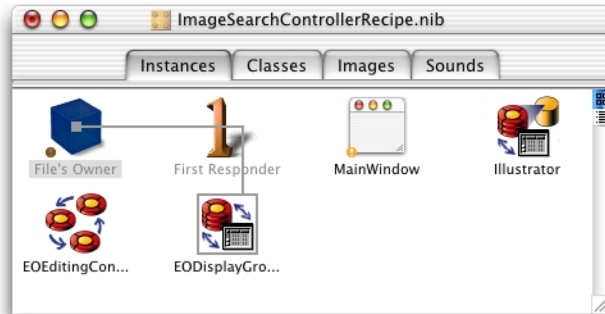
© Apple Computer, Inc. May 2002

Figure 19-5 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 19-6.

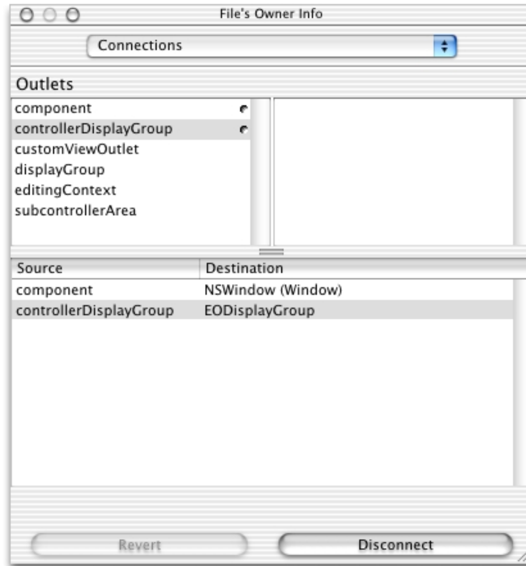
Figure 19-6 Bind File's Owner's `controllerDisplayGroup` outlet



Then, in the Info window, select `controllerDisplayGroup` and click Connect, as shown in Figure 19-7.

© Apple Computer, Inc. May 2002

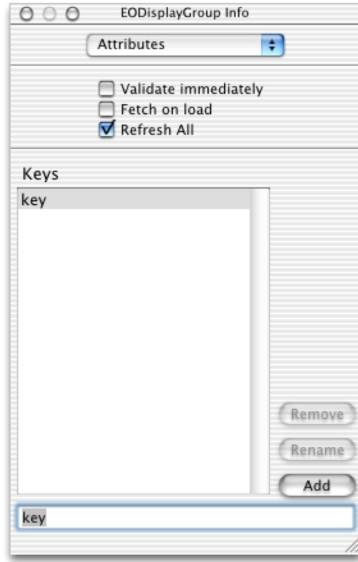
Figure 19-7 Bind the outlet



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 19-8 (page 281).

© Apple Computer, Inc. May 2002

Figure 19-8 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 19-9.

© Apple Computer, Inc. May 2002

Figure 19-9 Bind selectedIndex attribute of association to display group key

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.

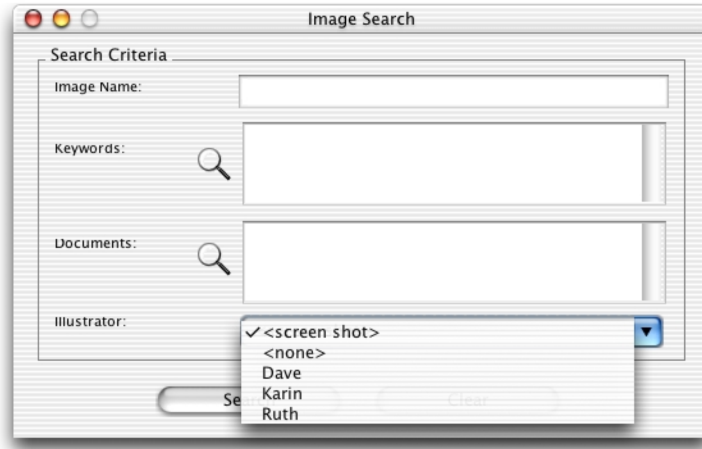
```
public void setKey(int illustrator) {
     _illustrator =
     (String)controllerDisplayGroup().valueForObjectAtIndex(illustrator,
       "name");
}
```

It sets an instance variable in the class (`_illustrator`) to the String value of the object selected in the menu.

Figure 19-10 shows a pop-up menu in action.

© Apple Computer, Inc. May 2002

Figure 19-10 A pop-up menu in action



© Apple Computer, Inc. May 2002

© Apple Computer, Inc. May 2002

Task: Building a Login Window

Whether you know it or not, after reading through the other tasks, you know all you need to know to implement a login window in a Direct to Java Client application. This task provides a road map and some helpful hints to successfully implement a login window in your application.

The DiscussionBoard application that's included with WebObjects 5.1 provides a login window based solely on rules and custom controller classes. It is a fine example but there are easier ways to accomplish the same thing, as this chapter describes.

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 20-1.

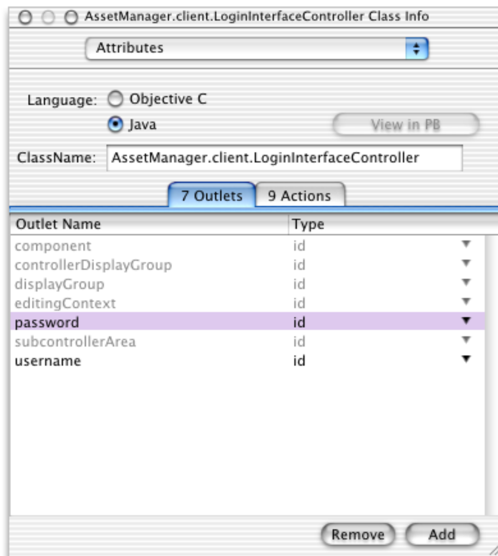
© Apple Computer, Inc. May 2002

Figure 20-1 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 245) 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 245) or "Programmatic Access to Interface Components" (page 185) to learn how to add outlets. The outlets pane should then appear as shown in Figure 20-2.

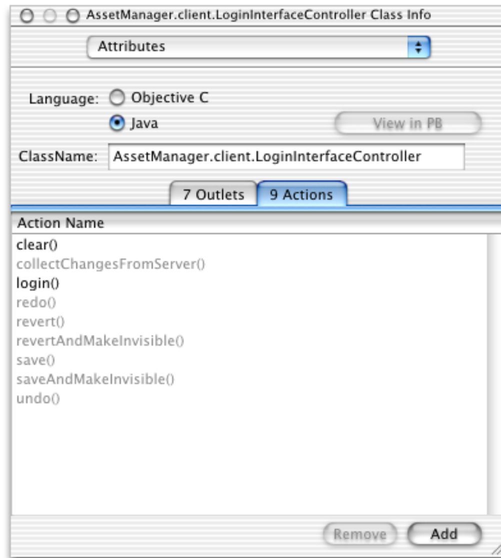
Figure 20-2 Add outlets named username and password



© Apple Computer, Inc. May 2002

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 20-3.

Figure 20-3 Add actions



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. File's Owner's connections should then appear as shown in Figure 20-4.

© Apple Computer, Inc. May 2002

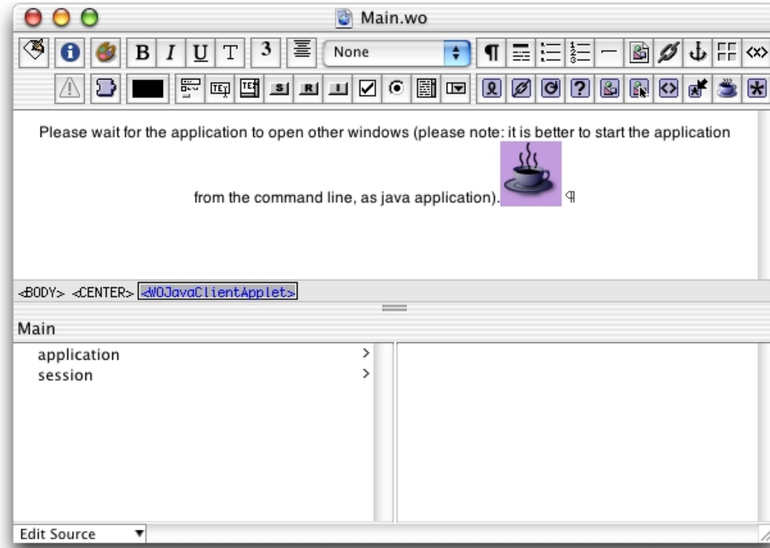
Figure 20-4 File's Owner with new connections

Save the nib file.

To load this interface when the application launches, you use a binding provided on the Main component. Open `Main.wo` in WebObjects Builder and select the WOJavaClientApplet dynamic element, as shown in Figure 20-5.

© Apple Computer, Inc. May 2002

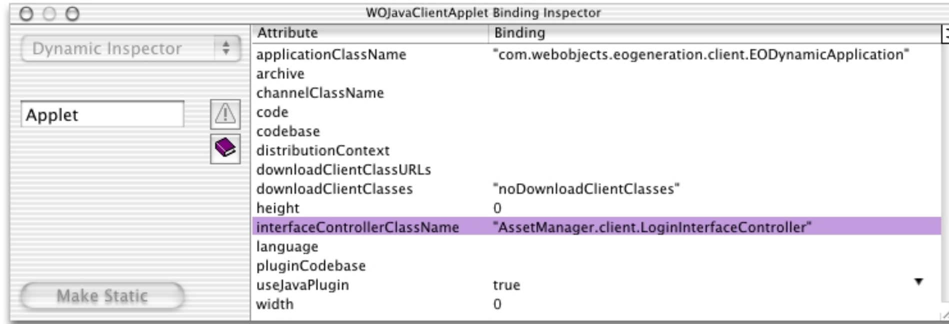
Figure 20-5 Select the WOJavaClientApplet dynamic element



Then, open the WOJavaClientApplet Binding Inspector by choosing Inspector from the Window menu. This shows you all the possible bindings for the WOJavaClientApplet 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 20-6.

© Apple Computer, Inc. May 2002

Figure 20-6 Add value for `interfaceControllerClassName` binding



The nib file specified as the value for this binding is loaded when the application starts up. Save the Main component and build and run the project and the login nib file should appear.

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 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 185). 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.

```
public EOTextField username, password;
```

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() {}
```


© Apple Computer, Inc. May 2002

The `clear` method simply clears the values of the text fields. Add this code to it:

```
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 20-1 to the nib file's controller class to perform the remote method invocation. If the user successfully authenticates, the method returns `true`.

Listing 20-1 Client-side login method

```
public boolean clientSideRequestLogin() {
    EOGlobalID person =
         (EOGlobalID)(_distributedObjectStore().invokeStatelessRemoteMethodWithKeyPath(
         "session", "clientSideRequestLogin", new Class[] {String.class, String.class},
          new Object[] {username.getText(), password.getText()}));
    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 112) since the remote method invocation is invoked on the client's parent object store.

Now, invoke the client-side method `clientSideRequestLogin` in the `login` method, adding a conditional based on the response, as shown in Listing 20-2 (page 292).

© Apple Computer, Inc. May 2002

Listing 20-2 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 keypath "session" and the method `clientSideRequestLogin`, so add a method in `Session.java` with that name, as shown in Listing 20-3.

Listing 20-3 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, "userPassword");

      NSArray foundObjects = EOUtilities.objectsMatchingValues(editingContext, "Person",
        userCredentials);
        if (foundObjects.count() == 1) {

            user = (EOGenericRecord)foundObjects.objectAtIndex(0);

            return(editingContext.globalIDForObject(user));
        }
        else {
            return null;
        }
    }
```


© Apple Computer, Inc. May 2002

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 20-7.

Figure 20-7 Login failed

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.

© Apple Computer, Inc. May 2002

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 199). To learn how to suppress the default query window, see "The Default Query Window" (page 200).

Finally, now that you've disabled all the default mechanisms for users to use the application, you need to provide custom access. In Listing 20-2 (page 292), 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 ("Task: Using the Controller Factory Programmatically" (page 205)) 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 20-2 (page 292):

```
EOControllerFactory.sharedControllerFactory().queryControllerWithEntity
(<entity name>)
```

Make sure to add the import statement for the `com.webobjects.eogeneration.client` package to the nib file.

To load a nib file programmatically, change the `login` method in Listing 20-2 (page 292) to Listing 20-4.

© Apple Computer, Inc. May 2002

Listing 20-4 Load a nib file programmatically

```
public void login() {
     if (this.clientSideRequestLogin()) {
        MainMenuInterfaceController mainMenu = new
            MainMenuInterfaceController();                               //1
        EOFrameController.runControllerInNewFrame(mainMenu, null);       //2
     }
     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.

© Apple Computer, Inc. May 2002

© Apple Computer, Inc. May 2002

A P P E N D I X   A

XML Description of Classes and Actions

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 297) 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`

© Apple Computer, Inc. May 2002

`RaisedBezel`

`LoweredBezel`

`LineBorder`

editability

`Never`

`Always`

`IfSuperController`

alignment

`Center`

`Left`

`Right`

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".

© Apple Computer, Inc. May 2002

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*".

Classes With XML Tags and XML Attributes

Direct to Java Client user interfaces are defined in XML descriptions. This section lists all the classes in the Java Client frameworks that have at least one XML tag or one XML attribute. Classes (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`)

© Apple Computer, Inc. May 2002

com.webobjects.eogeneration.client.EOActionController

Superclass: `com.webobjects.eogeneration.client.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)

`titlesEntity` (`String` representing the names of the titles entity)

`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.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.EOAppletController

Superclass: `com.webobjects.eoapplication.EOComponentController`

Description: Represents applets like a window in an application; use only if running as an applet or in a browser.

XML Tag: None (never created from XML description)

XML Attributes: None

© Apple Computer, Inc. May 2002

com.webobjects.eoapplication.EOApplication

Superclass: `com.webobjects.eoapplication.EOController`

Description: A shared instance, manages global actions such as Quit and Save; includes API to work with documents.

XML Tag: None (never created from XML description)

XML Attributes: None

com.webobjects.eogeneration.client.EOAssociationController

Superclass: `com.webobjects.eogeneration.client.EOWidgetController`

Description: Handles associations for widgets, including the editable state and enabled key.

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)

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`)
`horizontalBorder` (`int`)
`titlePosition` (position)
`usesTitledBorder` (`boolean`)
`verticalBorder` (`int`)

© Apple Computer, Inc. May 2002

com.webobjects.eogeneration.client.EOCheckBoxController

Superclass: `com.webobjects.eogeneration.client.EOValueController`

Description: Handles checkboxes.

XML Tag: `CHECKBOXCONTROLLER`

XML Attributes: `displaysLabelInWidget` (`boolean`)

com.webobjects.eogeneration.client.EOComboBoxController

Superclass: `com.webobjects.eogeneration.client.EOTitlesController`

Description: Handles combo boxes with fixed values.

XML Tag: `COMBOBOXCONTROLLER`

XML Attributes: `isQueryWidget` (`boolean`)

`titlesEntity` (`String` representing the names of the titles entity)

`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: `alignmentWidth` (`int`)

`alignsComponents` (`boolean`)

`horizontallyResizable` (`boolean`)

`iconName` (`String`)

`iconURL` (`String`)

`label` (`String`)

`minimumHeight` (`int`)

`minimumWidth` (`int`)

`prefersIconOnly` (`boolean`)

`usesHorizontalLayout` (`boolean`)

`verticallyResizable` (`boolean`)

© Apple Computer, Inc. May 2002

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: `className` (String)

`disabledActionNames`(array of strings with names of methods to be disabled)

`transient` (`boolean`)

`typeName` (`String`)

com.webobjects.eoapplication.EODialogController

Superclass: `com.webobjects.eoapplication.EOSimpleWindowController`

Description: Handles dialogs (such as error messages).

XML Tag: `DIALOGCONTROLLER`

com.webobjects.eoapplication.EODocumentController

Superclass: `com.webobjects.eoapplication.EOEntityController`

Description: Handles editable documents.

XML Tag: `DOCUMENTCONTROLLER`

XML Attributes: `editability` (editability)

com.webobjects.eogeneration.client.EODynamicApplication

Superclass: `com.webobjects.eogeneration.client.EOApplication`

Description: Warms up controller factory and manages special entry actions.

XML Tag: None (never created from XML description)

XML Attributes: None

© Apple Computer, Inc. May 2002

com.webobjects.eogeneration.client.EOEditingController

Superclass: `com.webobjects.eoapplication.EODocumentController`

Description: Handles master-detail associations.

XML Tag: None

XML Attributes: None

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)
`archive` (`String` representing the name of the interface file to be used)
`displayGroupProviderMethodName` (method name)
`editingContextProviderMethodName` (method name)

com.webobjects.eogeneration.client.EOEnumerationController

Superclass: `com.webobjects.eogeneration.client.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.client.EOFormatValueController

Superclass: `com.webobjects.eogeneration.client.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)

© Apple Computer, Inc. May 2002

com.webobjects.eogeneration.client.EOFormController

Superclass: `com.webobjects.eogeneration.client.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.client.EOImageViewController

Superclass: `com.webobjects.eogeneration.client.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`

© Apple Computer, Inc. May 2002

com.webobjects.eogeneration.client.EOListController

Superclass: `com.webobjects.eogeneration.client.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.client.EOMultipleValuesEnumerationController

Superclass: `com.webobjects.eogeneration.client.EOEnumerationController`

Description: Handles to-many relationships to enumeration entities.

XML Tag: `MULTIPLEVALUESENUMERATIONCONTROLLER`

XML Attributes: `usesTableLabels` (`boolean`)

com.webobjects.eogeneration.client.EOOneValueEnumerationController

Superclass: `com.webobjects.eogeneration.client.EOEnumerationController`

Description: Handles to-one relationships to enumeration entities.

XML Tag: `ONEVALUEENUMERATIONCONTROLLER`

XML Attributes: `isQueryWidget` (`boolean`)

© Apple Computer, Inc. May 2002

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.client.EOQueryController

Superclass: `com.webobjects.eoapplication.EOEntityController`

Description: Handles query interfaces.

XML Tag: `QUERYCONTROLLER`

XML Attributes: `editability` (editability)
`runsConfirmDialogForEmptyQualifiers` (`boolean`)

com.webobjects.eogeneration.client.EOQuickTimeViewController

Superclass: `com.webobjects.eogeneration.client.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.client.EORangeTextFieldController

Superclass: `com.webobjects.eogeneration.client.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`)

© Apple Computer, Inc. May 2002

com.webobjects.eogeneration.client.EORangeValueController

Superclass: `com.webobjects.eogeneration.client.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).

com.webobjects.eogeneration.client.EORangeWidgetController

Superclass: `com.webobjects.eogeneration.client.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.eogeneration.client.EOStaticIconController

Superclass: `com.webobjects.eoapplication.EOComponentController`

Description: Handles the display of static icons.

XML Tag: `STATICICONCONTROLLER`

© Apple Computer, Inc. May 2002

com.webobjects.eogeneration.client.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.client.EOStaticTextFieldController

Superclass: `com.webobjects.eogeneration.client.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.client.EOTableColumnController

Superclass: `com.webobjects.eogeneration.client.EOFormatValueController`

Description: Handles table columns.

XML Tag: `TABLECOLUMNCONTROLLER`

com.webobjects.eogeneration.client.EOTableController

Superclass: `com.webobjects.eogeneration.client.EOAssociationController`

Description: Handles table views.

© Apple Computer, Inc. May 2002

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.client.EOTextAreaController

Superclass: com.webobjects.eogeneration.client.
EOValueAndURLController

Description: Handles scrollable text areas.

XML Tag: TEXTAREACONTROLLER

com.webobjects.eogeneration.client.EOTextFieldController

Superclass: com.webobjects.eogeneration.client.
EOFormatValueController

Description: Handles editable text fields.

XML Tag: TEXTFIELDCONTROLLER

XML Attributes: isQueryWidget (boolean)

com.webobjects.eogeneration.client.EOTitlesController

Superclass: com.webobjects.eogeneration.client.
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: titleKeys (array of strings represent the key paths to be displayed for title objects)
titlesDisplayGroupProviderMethodName (method name)

© Apple Computer, Inc. May 2002

com.webobjects.eogeneration.client.EOValueAndURLController

Superclass: `com.webobjects.eogeneration.client.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.client.EOValueController

Superclass: `com.webobjects.eogeneration.client.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.client.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)
`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)

© Apple Computer, Inc. May 2002

XML Attributes: `windowPosition` (position)
`usesAction` (`boolean`)
`usesButton` (`boolean`)
`usesUserDefaultsWindowLocation` (`boolean`)
`usesUserDefaultsWindowSize` (`boolean`)

EOActions XML Descriptions

APPLICATIONACTION

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`)

CONTROLLERHIERARCHYACTION

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`)

© Apple Computer, Inc. May 2002

HELPWINDOWACTION

XML Attributes:
`shortDescription` (`String`)
`menuAccelerator` (`String`—example "shift P")
`multipleWindowsAvailable` (`boolean`)
`task` (`String`)

INSERTACTION

XML Attributes: `entity` (`String`)

OPENACTION

XML Attributes: `entity` (`String`)

QUERYACTION

XML Attributes: `entity` (`String`)

STANDARDACTION

XML Attributes: `entity` (`String`)

TOOLWINDOWACTION

XML Attributes:
`shortDescription` (`String`)
`menuAccelerator` (`String`—example "shift P")
`multipleWindowsAvailable` (`boolean`)
`task` (`String`)

WINDOWACTION

XML Attributes:
`actionPriority` (`int`)
`categoryPriority` (`int`)
`categoryName` (`String`)
`descriptionPath` (`String`)
`iconName` (`String`)
`iconURL` (`String`)
`menuAccelerator` (`String`—example "shift P")
`multipleWindowsAvailable` (`boolean`)

© Apple Computer, Inc. May 2002

`shortDescription` (String)
`smallIconName` (String)
`smallIconURL` (String)
`task` (String)

© Apple Computer, Inc. May 2002

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

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

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

Cocoa A framework containing all the objects you need to implement your graphical, event-driven user interface: windows, dialogs, buttons, menus, scroll bars, and text fields. Cocoa handles all the details for you as it efficiently draws on the screen, communicates with hardware devices and screen buffers, clears areas of the screen before drawing, and clips views.

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.

© Apple Computer, Inc. May 2002

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

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

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

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

Direct to Web A WebObjects development approach that can generate a HTML-based Web 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 your component.

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

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

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

EOModeler A tool used to create and edit models.

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

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

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

© Apple Computer, Inc. May 2002

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

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

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

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

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

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

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

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

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

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

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

key-value pair See key-value coding.

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

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

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

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

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

© Apple Computer, Inc. May 2002

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

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

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

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

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

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

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

referential integrity The rules governing the consistency of relationships.

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

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

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

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

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

© Apple Computer, Inc. May 2002

response A message conforming to the Hypertext Transfer Protocol (HTTP) sent from the Web server to the user's Web browser that contains the resource specified by the corresponding request. The response is typically a Web page. See also request.

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

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

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

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

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

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

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

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

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

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

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

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

WebObjects Builder A tool used to graphically edit WebObjects components.

© Apple Computer, Inc. May 2002

© Apple Computer, Inc. May 2002

Index

A

access layer
  abstract nature 41
  essential classes 41
  overview 41
  primary keys 60
  responsibilities 41
access layer classes
  EOAdaptor 42
    isolation from any particular data store 42
  EODatabaseContext 42
  EOModel 42
  EOUtilities 42
    appropriate usage 42
actions
  adding to Direct to Java Client applications 148
  adding with Assistant 153
  adding with custom controller class 237
  adding with XML tags 237
additionalAvailableSpecifications 199
appletviewer 80
application layer
  overview 50
Assistant
  additional property key path 98
  available in rapid turnaround mode 94
  customization costs 120
  disabling 94
  entities 94
  inside view 95
  miscellaneous tab 99
  properties tab 91, 96
  widgets tab 92, 99
  windows tab 92, 99
  XML tab 92, 99
associations
  examples 49
  in interface layer 48
  overview 48
  types 49
audience 20
awakeFromClientUpdate 113
awakeFromInsertion
  implementation 142

B

business logic 39
  abstraction from data stores 39
  adding to model 128
  adding to project 129
  custom classes 60
  custom classes and targets 132
  custom code 113, 138
    awakeFromClientUpdate 113
    invocations 139
    prepareValuesForClient 113
  initial values 142
  reusability 39
  schema information guidelines 39
  targets for custom classes 132
  validation 140
  values in enterprise objects 113
business logic partitioning
  common superclass 108
  design recommendations 108
  introduction to 107
  key to 40
  objects in 114
  or object distribution 23
  parameters of 40
  performance considerations 109
  remote method invocations 110
by-copy distribution mechanism 39

© Apple Computer, Inc. May 2002

C

class properties
  client-side 66
    display in client application 85
    relationships 103
  in EOModeler 66
client application
  application startup 83
  running 80
    applicationURL argument 82
    classpath argument 82
    client launch script 81
    from command line 82
    java command 82
    WOAutoOpenClientApplication 81
    wojavaclient.jar file 82
  using 84
client interface layer
  overview 48
client JDBC
  architectural comparison 33
client-side class properties 85
  display in client application 85
comparison of the two development approaches 31
confirmation dialog
  in query windows 147
considerations when choosing Java Client 25
  client-side processing 26
  network environment 26
  performance 26
  portability 26
  security 26
  system administration 26
  user experience 27
control layer
  essential classes 43
  overview 43
  same on client and server 43
control layer classes
  EOCustomObject 44
  EOEditingContext 44
  EOEnterpriseObject 43
  EOFetchSpecification 46
  EOGenericRecord 44
  EOGlobalID 46
  EOQualifier 46
  EOTemporaryGlobalID 46
controller factory
  using programmatically 161
controller hierarchy
  creation of 145
  overview 144
controllers
  application level 144
  entity level 145
  overview 144
  property level 145
  user interface level 145
CORBA
  as a distribution channel protocol 47
custom controller class
  writing 237
custom interfaces
  adding actions 179
  Cocoa to Swing translation 187
  integrating
    class association 173, 242
  integrating in Direct applications
    rule to integrate 243
  integrating into Direct applications 217, 253
    suppress subcontroller generation 244
  master-detail interface 181
  programmatic access to 185
  typical 29
custom rule
  for confirmation dialog 147
customization
  costs
    Assistant 120
    controller classes 121
    freezing interface files 121
    freezing XML 121
    rules 120
  tools
    Assistant 88, 119
    controller classes 121
    freezing XML 121
    frozen interface files 121
    rules 120
customizing client application 88
customizing frozen XML 234

© Apple Computer, Inc. May 2002

D

D2WComponents
  in the rule system 191
d2wmodels 231
  merged at run time 231
data stores
  connectivity options 37, 59
debugging
  D2WTraceRuleFiringEnabled 165
  how to 165
defaultAvailableSpecifications 199
Direct to Java Client applications
  adding actions 148
    Assistant 153
    custom controller class 237
    edit XML 237
  customization techniques 28
  customizing 119
    Assistant 119
    controller class 121
    freezing XML 121
    frozen interface files 121
    rules 120
  integrating custom interfaces 217, 253
display groups
  in interface layer 48
  overview 48
distribution layer
  data synchronization 116
  delegates 117
    for encryption and decryption 118
  distribution channels 117
  essential classes 47
  flow 114
  objects 114
  overview 46
  server-side 47
  transport protocol 117
distribution layer classes
  EODistributedDataSource 47
  EODistributedObjectStore 47, 115
  EODistributionChannel 47, 115
  EODistributionContext 47, 116
  EOHTTPChannel 47
  WOJavaClientApplet 115
documentation
  reader road map 21
  related documents 22
dynamically-generated interface
  typical 29

E

editing contexts
  synchronization 116
Enterprise Object
  definition 36
enterprise object
  class implementation 44
Enterprise Objects
  built-in features 85
  concrete implementation 44
  custom classes 60
  data access technology 32
  encyrption of 118
  layers 35
  models 37
  object graph
    synchronization 116
  responsibilities 32
  role in WebObjects 35
EnterpriseObjects palette 169
  loading 170
entities
  enumeration 95
  in rule system 95
  main 95
  other 96
  rule system definition 94
entity-relationship 37
EOAdaptorDebugEnabled 165
EOClassDescription, role with enterprise objects 36
EODatabaseContext
  role on server 44
EODistributedObjectStore
  role on client 44

© Apple Computer, Inc. May 2002

EODistributionContext 47
EOModel
  synchronizing with schema 86
  updating 86
EOModeler
  attribute properties 65
  business logic 128
  class properties
    client-side 66
  generate Java business logic classes 130
  model from data store
    "Ask about relationships" 60
    "Ask about stored procedures" 60
    "Assign primary keys to all entities" 60
    "Use Custom Enterprise objects" 60
  overview 57
  reverse-engineering schema 59
  SQL generation 67
  SQL generation options
    Create Database 68
    Create Primary Key Support 68
    Create Tables 68
    Drop Database 68
    Drop Primary Key Support 68
    Drop Tables 68
    Foreign Key Constraints 68
    Primary Key Constraints 68
EOObjectStoreCoordinator, overview of 46
EOSwitchComponent 196

F

faulting
  overview 44
File's Owner
  class association for integrated custom interfaces 173, 242
foreign key
  adding to entity 100
formatters
  apply to widgets 136
  in custom interfaces 177
Foundation classes
  NSArray 41
  NSBundle 41
  NSDictionary 41
  NSKeyValueCoding 41
  NSLog 41
Foundation framework
  essential classes 40
  in WebObjects 40
  overview 40
  versus JDK foundation classes 40
freezing XML
  customization costs 121
  customizing XML 234
  how to 227
  when to 228

G, H

generateSubcontrollers 244
generation layer
  overview 50
  using API programmatically 161

I

Inside WebObjects: Deploying WebObjects Applications 80
integrate custom interfaces into Direct applications 28
Interface Builder
  and EOModeler 174
  Cocoa to Swing translation 187
  EnterpriseObjects palette 169
  formatters 177
  laying out user interface 171
interface layer
  associations 48
  display groups 48
  overview 48
invokeRemoteMethod
  on EOCustomObject 110

© Apple Computer, Inc. May 2002

invokeRemoteMethodOnKeyPath
  on EODistributedObjectStore 111
invokeStatelessRemoteMethodWithKeyPath
  business logic 112
  editing contexts 112
  on EODistributedObjectStore 111

J

Java Client
  considerations when choosing 25
  definition 19
  deployment options 24
  features 22
  rapid application development 24
  rich user interface 22
  when to use 24
Java Client applications
  client-server synchronization 116
  customizing 88
  debugging 165
  deploying
    appletviewer 80
  mixing interface types 217, 253
  nondirect development 167
  running 80
Java Client architecture
  advantages 33
  compare to others 32
  compared to client JDBC 33
  compared to JDBC three-tier 33
  overview 37
  versus client stub design 39
Java Client Class Loader
  bindings on WOJavaClientApplet 72
  configurating 72
  options 72
  to ease deployment 25
Java Client deployment
  application upgrades 53
  installation 53
  issues 53
  performance 54
  platform support 53
  security 54
  server requirements 54
  user experience 54
Java Client development
  choosing an approach 30
  mixing the two approaches 28
  suggested approach 28
  suggested approach to 21
  two approaches to 27
Java Foundation Classes 22
Java Runtime Environment
  client requirement 26
Java Virtual Machine 19
JDBC
  compatibility 37
  connectivity
    compatibility 59
  three-tier
    architectural comparison 33
JNDI
  as a data access mechanism 35
  connectivity in WebObjects 5.1 59

K

key-value coding
  access mechanism for enterprise objects 43

L

launch arguments
  server
    for development 80
    WOAutoOpenClientApplication 80
    WOAutoOpenInBrowser 81
    WOPort 81
  session timeout 77

© Apple Computer, Inc. May 2002

M

model
  adding custom business logic 128
  attributes
    naming conventions 63
  entities
    naming conventions 63
  files 37
model files
  schema information stored in 37
Model-View-Controller paradigm
  overview 52
  use of "model" 52

N

nondirect development
  disadvantages 31
  how to 167

O

object distribution 23
object-relational mapping 37
  model files 37
OpenBase
  included database software 56
OpenBase Manager
  verifying schema synchronization 87

P

prepareValuesForClient 113
primary keys
  appropriate usage 60
  in access layer 60
  purpose 66
Project Builder
  classes 74
  Developer Help Center 22
  project
    default 73
Project Builder project
  adding business logic 129
  client files 75
  documentation 74
  frameworks 74
  groups 73
  interfaces 74
  products 74
  resources 74
  server files 75
  targets 74, 133
    build-style 75
    root-style 75
  web components 74
  web server resources 74
property keys 98
  adding with Assistant 98

Q

query windows
  confirmation dialog 147
questions
  in rule system 98

R

rapid application development 24
rapid turnaround mode 94
relationship manipulation
  in enterprise objects 44
relationships
  adding to model 99
  as client-side class properties 103
  making in EOModeler 101
remote method invocation
  "clientSideRequest" 111
  editing contexts 111
    controlling push to server 111
    synchronization 111

© Apple Computer, Inc. May 2002

remote method invocation (*continued*)
on application logic 111
on business logic 110
on EODistributedObjectStore 111
overview 110
security considerations 111, 112
reverse-engineering
in EOModeler 59
rule system
custom rules 146
d2wmodel files 231
debugging 165
flow 191
how it works 189
priorities 191
requests 191
trace firing of rules
debugging 165
rules
customization costs 120
priorities 191
running applications
as applets
considerations 25
client application 80
running as desktop application
special requirements 25

S

schema synchronization 86
when it's necessary 88
session timeout
adjusting for Java Client applications 77
in deployment environment 79
in Java Client applications 79
specifications in the rule system
definition 199
SQL generation
after adding entity 101
EOModeler 67
Swing 19
Swing toolkit 22

T, U

targets
definition 74
pop-up menu 75
three types 133
tasks
defined by rule system 96
form 96
identify 97
list 97
query 97
transport protocol
HTTP 26
others 26

V

validation
implementations 140
in client classes 141
in server classes 141
overview 44
where to validate 141

W

WebObjects
basic features 23
WebObjects CD-ROM 22
WOJavaClientApplet
bindings
Java Client Class Loader 72
WOJavaClientApplet, overview 47
WOXMLNode 196

X, Y, Z

XML tab in Assistant 92

© Apple Computer, Inc. May 2002

© Apple Computer, Inc. May 2002