GETTING STARTED WITH DIRECT TO JAVA CLIENT

Draft
12/21/99

This manual describes the Direct to Java Client feature of WebObjects 4.5.

Contents

Table of Contents

Creating a Direct to Java Client Project 13
- What's in the Template Project? 15

Building and Running the Application 15
- If the Client Application Doesn't Start 17
- If the Application Has No Windows 17

Examining the Application 18
- Main, Enumeration, and "Other" Entities 21
  - Main Entities 21
  - Enumeration Entities 21
  - "Other" Entities 22
- Customizing the Application 22

The Assistant 23
- If the Assistant Command Isn't Available 23
- Disabling the Assistant 24

Configuring Entities 25

Configuring Properties 26

Configuring Widgets 28

Typical Workflow 30
- Examining the Panes of the Query Window 31
  - Configuring the Studio Pane of the Query Window 31
  - Configuring the Customer Pane of the Query Window 31
  - Configuring the Unit Pane of the Query Window 32
  - Configuring the Video Pane of the Query Window 33
- Examining Form Windows 33
  - Configuring the Customer Form Window 34
  - Configuring the Unit Form Window 35

Configuring Windows 36
- Changing the Title of the Query Window 36

Other Assistant Settings 37
Advantages of Direct to Java Client 37
To Do 38
Architectural Overview 41
Controller Hierarchy 43
Controllers 44
Creating the Controller Hierarchy 44
Unarchiving XML 45
Server Side XML Generation 47
The Rule System 47
D2WComponents 48
Rule System Requests 49
Internal Rule System Requests 50
Generating the Select Studio Dialog 50
Customization Approaches 57
Other Approaches 59
Writing Custom Rules 59
Using the Rule Editor 60
When to Write Custom Rules 62
Trouble Shooting Custom Rules 63
Freezing XML 63
Creating a D2WComponent 64
Getting the Default XML 65
Modifying the XML 66
Writing a Custom Rule to Use Your Component 68
Freezing Nib Files 69
Adding Custom Actions 70
Adding an Additional Action 70
Creating a Corresponding Window 71
(Optional) Specifying Additional Available Specifications 72
XML Tags and Attributes for EOActions 73
Implementing Custom Controller Classes 74

Preface

Direct to Java Client is an addition to Enterprise Objects Framework and the Java Client technology that dynamically generates user interfaces for Java Client applications.

This book provides an overview of the applications you can create with Direct to Java Client. It takes you step by step through the process of creating, building, and running a Direct to Java Client application, as well as through the process of customizing an application using the Direct to Java Client Assistant.

Before you begin doing this tutorial, you should be familiar with the Java Client technology. If you're not, work through the *Java Client Tutorial* to acquaint yourself.

This book is organized into three chapters:

- "Getting Started with Direct to Java Client" (page 11), teaches you how to create a Direct to Java Client project, how to build and run your application, and how to use the Direct to Java Client Assistant to customize it.
- "Understanding Direct to Java Client Applications" (page 39), describes the architecture of Direct to Java Client and its fundamental concepts.
- "Customizing Direct to Java Client Applications" (page 55), explains the different approaches to customize Direct to Java Client applications beyond using the Assistant.

After you have read this book, you should have a good working knowledge of Direct to Java Client. For more in depth information on the Direct to Java Client classes and interfaces, see the *EOApplication Framework Reference* and the *EOGeneration Framework Reference*.

Chapter 1

Getting Started with Direct to Java Client

This chapter takes you step by step through the process of creating, building, and running a Direct to Java Client application, as well as through the process of customizing an application using the Direct to Java Client Assistant.

Note: There are other methods of customizing a Direct to Java Client application besides using the Assistant, but they are more advanced and are covered in later chapters.

Before you begin doing this tutorial, you should be familiar with the Java Client technology. If you're not, work through the *Java Client Tutorial* to acquaint yourself.

This tutorial uses a sample OpenBase Lite database, Movies, that contains information about movies and a video rental store. OpenBase Lite and the sample Movies database come with WebObjects and are used by the WebObjects and Enterprise Objects Framework examples. Neither requires any special installation or configuration.

Creating a Direct to Java Client Project

Direct to Java Client isn't yet fully integrated into Project Builder like other WebObjects technologies, so you don't create Direct to Java Client projects the same way you create other kinds of projects. Instead of creating a new project in Project Builder, you copy a template project, make a few modifications, and proceed from there as you normally would.

The following steps describe the process for the tutorial application.

1. Copy the DirectToJavaClientTemplate project directory to a working directory.

   The DirectToJavaClientTemplate project directory is located in **/System/Developer/Examples/WebObjects/JavaClient** (Mac OS X Server) or **$NEXT_ROOT/Developer/Examples/WebObjects/JavaClient** (Windows). Make a copy of the project in your development directory. This tutorial uses **/tmp** (MacOS X Server) or **C:/tmp** (Windows).

2. Open your copy of the project.

Navigate to your copy of the project directory, and open the **PB.project** file. This opens the project in Project Builder, starting Project Builder first if it isn't already running.

3. Change the name of your project to "D2JCTutorial".

   Open Project Builder's Project Attributes inspector: Choose Tools ▸ Inspector to open the inspector panel and choose Project Attributes in the upper pull-down list to switch to the Project Attributes view.

   Type `D2JCTutorial` in the Name field and press Return. Click Yes when Project Builder asks whether to rename the project folder too.

4. Rename the main file to **D2JCTutorial_main.m**.

   In the Other Sources bucket of your project, select the **DirectToJavaClientTemplate_main.m** file. Choose File ▸ Rename, and change the file's name to **D2JCTutorial_main.m**. (The main file name should be based on the project name: ***ProjectName*_main.m**.)

Direct to Java Client technology uses model files to generate applications. To make models available to Direct to Java Client, you add them to your project. You can either add models directly to your project by adding them to the Resources suitcase, or you can add them indirectly by adding frameworks that contain your models.

The models used by this tutorial application are the Movies and Rentals models that describe the Movies database. These models are contained in the BusinessLogicJava framework, which provides server side business logic implemented in custom enterprise objects. The corresponding client side objects are provided by the BusinessLogicClient framework, which this tutorial application also uses.

5. Add **BusinessLogicJava.framework** and **BusinessLogicClient.framework** to your project.

   Double click the Frameworks suitcase of your project. In the Add Frameworks panel, navigate to **/Local/Library/Frameworks** (Mac OS X Server) or **$(NEXT_ROOT)/Local/Library/Frameworks** (Windows). Choose **BusinessLogicJava.framework**, and click OK. Click Add when Project Builder asks whether to add the **Local/Library/Frameworks** directory to your Framework Search Order.

Repeat to add **BusinessLogicClient.framework**, which is in the same directory.

What's in the Template Project?

The DirectToJavaClientTemplate project is a normal Java Client project. It has the typical Application, Session, and Main classes for the server side application. The client side subproject, however, is empty. It has no interface controller or nib files, because Direct to Java Client application's generate their user interfaces dynamically.

The project's **Main.wo** contains a WOJavaClientApplet which is configured in a special way. It's **width** and **height** are set to 0, and the **applicationClassName** is set to "com.apple.client.eogeneration.EODynamicApplication". The application class is important because it's responsible for initiating the user interface generation process.

The Resources suitcase of the project contains a **user.d2wmodel** file. The Direct to Java Client Assistant uses this file to store user interface configuration information.

Building and Running the Application

Building and running the application is just like building and running any other Java Client application. Simply perform the following steps.

1. Build the application.

   Choose Tools ▸ Project Build ▸ Build. This opens the Build Project panel and builds the application.

2. Configure the application's launch options.

   Choose Tools ▸ Launcher ▸ Launcher. This opens the Launch panel. Click the check mark button to open the Launch Options panel. In the lower half of the window, ensure that the Arguments tab is selected. Click Add. Type the following, substituting values appropriate for your environment:

   On MacOS X Server:

```
-WOPort 8888 -WOAutoOpenInBrowser NO -NSProjectSearchPath '(/
tmp)'
```

On Windows:

```
-WOPort 8888 -WOAutoOpenInBrowser NO -NSProjectSearchPath '(C:/
tmp)'
```

The launch arguments are used as follows:

- *WOPort*. Specify any port you want. It is not necessary to specify a port, but it is convenient. By specifying a port, the application URL (used to start the client application) doesn't change. If the application URL is always the same, you can start the client application with the same command each time you run it.
- *WOAutoOpenInBrowser*. Specify NO so the application is not automatically launched in a browser. Java Client applications should be run as Java applications (as described in the next step).
- *NSProjectSearchPath*. Specify the directory in which your project is located. This enables Rapid Turnaround Mode. You must be in this mode to dynamically configure a Direct to Java Client application with the Assistant. This argument might not be necessary if your NSProjectSearchPath user default is set correctly (which it is by default). For more information, see "If the Assistant Command Isn't Available" (page 23).

3. Start the server application.

   On the Launch panel, click the Launch button (the left most button).

4. Start the client application.

   In a shell, start the client as an application.

   On MacOS X Server:

```
java -classpath "/Local/Library/Frameworks/
BusinessLogicClient.framework/WebServerResources/Java:/System/
Library/Java/eojavaclient.jar:/System/Library/Frameworks/
JavaVM.framework/Classes/classes.jar:/System/Library/
Frameworks/JavaVM.framework/Classes/awt.jar:/System/Library/
Frameworks/JavaVM.framework/Classes/swingall.jar"
com.apple.client.eoapplication.EOApplication -applicationURL
http://localhost:8888/cgi-bin/WebObjects/D2JCTutorial
```

   On Windows (substituting your $NEXT_ROOT if necessary):

```
java -classpath "C:\Apple\Local\Library\Frameworks\
BusinessLogicClient.framework\WebServerResources\Java;C:\Apple\
Library\Java\eojavaclient.jar;C:\Apple\Library\JDK\lib\
classes.zip;C:\Apple\Library\JDK\lib\swingall.jar"
com.apple.client.eoapplication.EOApplication -applicationURL
http://localhost:8888/cgi-bin/WebObjects/D2JCTutorial
```

Warning: Do not put `C:\Apple\Library\Frameworks\JavaVM.framework\Classes\awt.jar` in your classpath on Windows. It is required on MacOS X Server, but you shouldn't include it on other platforms.

Note: For the final deployment of the client application, you should build a jar file (or a small number jar files) containing the classes used by the application (classes in the Java Client packages, classes in swingall.jar, and your custom classes). Then to start the client application, you list only those jar files in the classpath.

If the Client Application Doesn't Start

Verify that your classpath is set correctly. Remember to quote the classpath argument. Be sure that you use the correct path separator; on MacOS X Server it's a colon (":"), while on Windows it's a semicolon (";").

Verify that your application URL is correct. Check the messages in the Launch panel. At the bottom, you'll see the following messages:

```
Sep 28 16:08:56 D2JCTutorial[679] Your application's URL is:
http://localhost:8888/cgi-bin/WebObjects/D2JCTutorial
Sep 28 16:08:56 D2JCTutorial[679] Waiting for requests...
```

Verify that the application URL you use to start the application is the same as the one in the Launch panel message.

If the Application Has No Windows

Verify that you've added the BusinessLogicJava framework to your project. Direct to Java Client dynamically generates an application based on model files. In this application, the model files come from the BusinessLogicJava framework.

Examining the Application

This section illustrates the application's functionality by guiding you through the steps to search for records, modify records, and add new ones.

When the application starts, it opens a window with which you can search for database records. Each of the tabs in the Query Window correspond to an entity in one of the application's models. Not all of the entities in the Movie and Rentals models have tabs in the Query Window. The way Direct to Java Client chooses which entities to represent in the Query Window is described in the section "Main, Enumeration, and "Other" Entities" (page 21).

1. Focus the Query Window on the Movie entity.

   In the Query Window, click the Movie tab. The Query Window switches its contents to a user interface for searching for Movies.

   Note that the default Direct to Java Client application allows you to search on **title** and **plotSummary**. You can easily configure the application to search on different attributes using the Direct to Java Client Assistant (Assistant for short). You will change the search attributes later in this tutorial.

2. Search for Movies whose titles begin with the letter "A".

   Type "A" in the Title field, and click the Find button. The application lists the Movies meeting the search criteria.

   Note that the application lists **title**, **category**, **dateReleased**, **posterName**, **revenue**, **trailerName**, **plotSummary** for the Movies in the result set. As with the search attributes, you can easily configure the application to display different attributes in the table, which you will do later.

3. Search for Movies whose titles begin with the letter "A" or "B".

   Click Clear. Although the "A" is cleared from the Name field, the Movies in the results table remain. To add Movies that start with the letter "B" to the results, type "B" in the Name field, and click Append. Now the results table contains Movies that start with the letter "A" and the letter "B".

   The difference between Find and Append is this: Find replaces the records in the results list with the results of the most recent search. Append adds data from the most recent search to the records already in the results

table. So using Append is a way to *OR* results together (for example, Movies whose titles start with "A" OR Movies whose titles start with "B").

4. Open a Movie.

   Select a Movie in the result list and click Open, or simply double-click the Movie's title. Either action opens the Movie in a separate window that displays its attributes and relationships. The attributes—**title**, **category**, **dateReleased**, **posterName**, **revenue**, and **trailerName**—are in the top part of the window. You can edit any of the Movie's attributes here. After making changes, you can save them (click Save at the top of the Movie window) or revert back to the original values (click Revert).

   The Movie's relationships are in the bottom part of the window. One relationship—**studios**—is displayed by itself in the middle of the window while the others are displayed below in a tab view. The reason for this is explained later in the section "Main, Enumeration, and "Other" Entities" (page 21).

   In the Studio area of the Movie window, you can change the Movie's Studio (Select), open the Movie's Studio (Open), or unassign the Movie's Studio (Deselect). Clicking the Open button simply opens the Movie's Studio in a Studio window which is similar to the Movie window in that it displays Studio attributes and relationships. Clicking Deselect unsets the Movie's **studio** relationship so that the Movie has no Studio.

5. Select a new Studio.

   In the Studio area of the Movie window, click Select. A dialog opens for searching for Studios. Type search criteria ("A" in the Name field, for example), and click Find. Choose a Studio from the result list and click OK. The dialog closes, and the Movie's Studio is changed from the original to the one you selected.

   Note that the Movie window now has an asterisk ("*") in the title bar. The asterisk indicates that the Movie has unsaved changes. If you wanted to save the change, you would click Save. In this case, however, click Revert to discard the change.

6. Examine the relationship tabs at the bottom of the window.

The relationships for the Plot Summary and Voting tabs—**plotSummary** and **voting**—are to-one relationships. Their respective panes allow you to modify the destination objects directly. For example, if you change the contents of the Summary field in the Plot Summary pane and click Save, you change the PlotSummary object's **summary** value and commit the change to the database.

The other relationships—**directors**, **reviews**, and **roles**—are to-many relationships. Their panes allow you to add destination objects to the relationships and to delete objects from them. Additionally you can open destination objects (in their own window) and edit them.

7. Create a new Movie.

   In the Movie window, click New. Alternatively, if the Query Window's Movie tab is still selected, you can click New in the Query Window. Either action creates a new Movie window into which you enter information about the new Movie. Fields with blue titles are required; that is, you can't save the new Movie if the blue titled fields are empty.

   Fill in the Movie's attributes as follows:

   - *Title:* Evil Dead *2*
   - *Category:* Horror
   - *Date Released:* 4/1/1987

   Click Save to save the new Movie.

8. Set the Movie's Studio to Elite Entertainment.

   In the Studio area of the window, click Select. In the Studio dialog, type "E" in the Name field, and click Find. No results are returned, so you'll have to create the Elite Entertainment Studio.

   Click New in the Studio dialog. The first dialog closes, and a new Studio dialog opens for creating a new Studio. Name the Studio "Elite Entertainment" and provide a budget. Click Save.

   The new Studio is saved to the database and assigned as the Studio for the "Evil Dead *2*" Movie.

   You can add directors and movie roles (and their actors) in a similar manner.

Main, Enumeration, and "Other" Entities

Direct to Java Client divides entities into three types: *main*, *enumeration*, and *other*. Each of an applications entities is exactly one of these types.

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. For example, in this tutorial application, the main entities are Customer, Movie, Studio, Talent, Unit, User, and Video.

Direct to Java Client defines a main entity as one that is not the destination of any relationships that:

- Propagate primary key
- Own destination
- Use the cascade delete rule

Enumeration Entities

Direct to Java Client also defines the concept of an *enumeration* entity. An enumeration entity is an entity that conforms to the conditions for main entities and additionally conforms to the following conditions:

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

In practice, enumeration entities should define a collection of values that represent a list of choices. For example, in this tutorial application, the enumeration entities are FeeTypes and RentalTerms.

The values in enumeration entities are usually fairly static, and you usually don't want a complex user interface for changing them. Consequently, Direct to Java Client applications provide a single window for editing enumeration values.

1. Open the Enumeration Window.

   Choose the "Enumeration Window" command in the Tools menu. This opens the Enumeration Window, which contains a tab for each of the application's enumeration entities. The tab for a particular entity shows the

complete set of values in that entity (which should be a small number of values). You can add a new value to the enumeration's collection (Add), delete a value from the collection (Remove), and modify a value (simply make the changes and click Save).

By displaying enumeration values in this special window, an application doesn't clutter the Query Window with tabs for enumeration entities. This approach simplifies the application's user interface in other ways, as well. The application doesn't provide form windows for enumeration entities, and relationships to enumeration entities can be represented simply with combo boxes (for to-one relationships) and pick lists (for to-many relationships) instead of with a tables (to-ones) and select dialogs (to-manys).

"Other" Entities

Entities that aren't main or enumeration entities are simply "other" entities. For example, in this tutorial application, the "other" entities are CreditCard, Director, Fee, MovieRole, PlotSummary, Rental, Review, TalentPhoto, and Voting.

"Other" entities can be manipulated through the master-detail user interfaces of main entities. This explains why a Movie's **studio** relationship is displayed by itself in the middle of the window while the other relationships are displayed below in a tab view. The destination entity of the **studio** relationship, Studio, is a main entity. Main entities have their own form windows with which you edit them. The destinations of the other relationships are "other" entities. Since they don't have their own form windows, they are edited in master-detail interfaces inside a main entity's form window.

Customizing the Application

As you can see, the default application is fairly complete. Direct to Java Client uses a sophisticated set of rules to assemble the user interface. However, the default user interface isn't exactly what you want in every case.

In the next sections you'll make changes to the application, including the following:

- Modify the set of main entities.

- Modify the search attributes in the Query Window, and specify the order in which the attributes are displayed. Similarly, modify the attributes displayed in the results table of the Query Window.
- Specify the order of the attributes and relationships that can be edited in a form.
- Change the widget used to display and edit particular attributes.

The Assistant

The Direct to Java Client Assistant is an easy-to-use tool that performs the most common customizations to Direct to Java Client applications. It runs inside the client application so you can see and directly test changes while the application is running.

1. Open the Assistant

   In any window of your application, choose Tools ▸ Assistant, which opens the Assistant window.

   The Tools menu might not list the Assistant command. If the application isn't running in Rapid Turnaround Mode, the Assistant can't be made available: see the section "If the Assistant Command Isn't Available" (page 23). Or the Assistant might be explicitly disabled: see "Disabling the Assistant" (page 24).

The Assistant window has several tabs across the top: Entities, Properties, Widgets, Windows, Miscellaneous, and XML. The tabs allow you to configure the application in different ways. Each is discussed in later sections.

If the Assistant Command Isn't Available

Verify that the Rapid Turnaround Mode is active. Check the messages in the Launch panel. At the bottom, you'll see the following messages:

```
Sep 28 16:08:56 D2JCTutorial[679] Your application's URL is:
http://localhost:8888/cgi-bin/WebObjects/D2JCTutorial
Sep 28 16:08:56 D2JCTutorial[679] Waiting for requests...
```

Look just above the "Your application's URL is..." message. If you see a message such as the following:

```
Sep 28 15:54:29 D2JCTutorial[666] ***** PLEASE NOTE: Rapid turn-
around is not active for your application. The configuration
assistant will not be made available in the client application since
it will not be able to modify your project's user.d2wmodel rule file.
Your NSProjectSearchPath user default must be set to an array of
paths where your project can be found, for example you can launch
your application like this: MyApp -NSProjectSearchPath '(/some/path/
theDirectoryWhereMyProjectIs)'. To completely disable the assistant
use a -EOAssistantEnabled NO flag when you start the application.
```

then your NSProjectSearchPath is set incorrectly.

Quit the client and server applications. Change the launch arguments for your application to include an argument for NSProjectSearchPath. (Refer to step *2* in "Building and Running the Application" on page 15 for help setting launch arguments.)

By default, NSProjectSearchPath is `../..`, which is generally sufficient for running "out of the box." However, if you've customized your build environment, you might need a different setting.When the argument is set properly, you should see the following message:

```
Sep 28 16:14:30 D2JCTutorial[683] ***** PLEASE NOTE: The
configuration assistant will be made available in the client
application. To disable the assistant use a -EOAssistantEnabled NO
flag when you start the application.
```

Disabling the Assistant

You should not deploy an application with the Assistant enabled. To disable it when you deploy, use the EOAssistantEnabled default, which enables and disables the Assistant. For example, to start the server application D2JCTutorial server application with the Assistant disabled, you could use the command:

On MacOS X Server:

```
D2JCTutorial -WOPort 8888 -WOAutoOpenInBrowser NO
-EOAssistantEnabled NO -NSProjectSearchPath '(/tmp)'
```

On Windows:

```
D2JCTutorial -WOPort 8888 -WOAutoOpenInBrowser NO
-EOAssistantEnabled NO -NSProjectSearchPath '(C:/tmp)'
```

Configuring Entities

The default set of rules that Direct to Java Client uses to identify main, enumeration, and "other" entities are a pretty good start. However, you might need to make some changes to the way Direct to Java Client classifies your application's entities. For example, this tutorial application doesn't use the User entity, so you need to remove it from the list of main entities.

1. Specify the User entity as an "Other" entity.

   In the Assistant window, ensure that the "Entity" tab is selected. In the Main Entities list, select User. Click the arrow pointing to the left to move the User entity from the Main Entities list to the Other Entities list.

2. Rearrange the entities.

   By default, the entities are ordered alphabetically. For example, in the Query Window, the Customer tab is first, then Movie, and so on. In this application, it makes more sense to group Movie, Talent, and Studio together and group Customer, Unit, and Video.

   In the Main Entities list, select Movie. Click the arrow pointing up to move the Movie entity to the top of the list. Move Talent so it's next in the list, then Studio. Customer, Unit, and Video, in that order, should be last in the list.

3. Apply the changes.

   Click Apply. This updates the Assistant's in-memory set of rules. The new rules don't affect existing windows, but they are applied when Direct to Java Client generates new user interfaces in the running client application.

   To see that your changes have been recorded in the Assistant's set of rules, choose "New Query Window" in the Tools menu. The new Query Window that opens doesn't have a tab for User, and the window's tabs are ordered as you specified in step 2.

4. Restart the application.

   Click Restart. This restarts the client application using the Assistant's in-memory set of rules. In the new instance of the application, the Query Window immediately displays the changes you specified.

Since you can't update an application's existing user interface by applying changes, you can use Restart to reflect your changes throughout the entire application. If you don't like the changes, you can revert them by clicking Revert.

5. Save your changes.

   Click Save. This updates the set of rules saved in your project. Once you have saved the changes, you can no longer revert.

Configuring Properties

The Assistant's Properties tab allows you to configure the way an application handles properties (attributes and relationships). For example, you can change the attributes that users search on in the Query Window and specify the order in which they appear.

1. Set the property keys for querying the Movie entity.

   In the Assistant, select the Properties tab. The specification settings—the Query, Task, and Entity pull-down lists—should be set to <All>, query, and Movie, respectively. The settings are probably correct, but if any of the settings are different, change them. A simple way to set the Task and Entity pull-down lists is to select the Movie tab in the Query Window. The Assistant automatically updates to set the Task pull-down list to query and the Entity pull-down list to Movie.

   Move **plotSummary** and **voting** to the Other Property Keys list. Move **category** and **studio** to Property Keys. Using the up and down arrows, order the properties as follows: **title**, **category**, **studio**.

   Save the changes, and restart the application (from the Assistant). The Movie pane of the Query Window is reconfigured.

   Note that a Studio Name field is now available. This is because you specified the **studio** relationship as a property key for the query task. When a to-one relationship (such as **studio**) is specified as a property key, Direct to Java Client uses *identify* property keys of the relationship's destination object to represent the relationship in the user interface.

2. Examine the identify property keys for the Studio entity.

   In the Assistant, set the Task pull-down list to identify and set the Entity pull-down list to Studio. Observe that the Studio has one identify property key—**name**. This attribute is used to "identify" a particular Studio when it appears in the user interface as a destination of a to-one relationship.

   One example of how the identify property keys are used is in the Movie pane of the Query Window. Movie's **studio** relationship is specified as a query property key. To represent **studio** as a property key in the Query Window, Direct to Java Client uses the identify property key (**name**) of the relationship's destination entity (Studio).

   Another example is the Studio part of a Movie window, in which the Movie's Studio is identified by its **name**.

3. Modify the identify property keys for the Movie entity.

   Set the Assistant's Entity pull-down list to Movie, keeping the Task as identify. Movie has one identify property key, **title**. Since it's quite common for movie remakes to use the same name as the original, a movie's title isn't always enough to identify it. Add **dateReleased** to Movie's list of identify properties, and save the change.

4. Configure the property keys for listing Movies.

   The property keys for the list task are used in result lists such as the ones in the Query Window. When you search for Movies, for example, the Movies in the result set are *listed* in the Query Window's table. The result table has a column for the list property keys, **title**, **category**, **dateReleased**, **posterName**, **revenue**, **trailerName**, and **plotSummary**. The columns for these properties are crowded into the table, and many aren't particularly helpful to users.

Note: In addition to the property keys listed above, **voting** is also a list property key for the Movie entity. The **voting** relationship doesn't have a column in the results table. This is because Voting's identify property key is it's **movie** relationship, which Direct to Java Client omits from the table view to avoid recursion.

   To pare down the set of columns, choose list in the Assistant's Task pull-down list, and choose Movie in the Entity pull-down list (if it isn't selected

already). Move **posterName**, **revenue**, **trailerName**, and **plotSummary** to the Other Properties column.

So users can verify that all the search criteria they specify is applied correctly, the list property keys should be a super set of the query properties. Since users can search for Movies by Studio, the list property keys should include **studio**, as well. Add **studio** to the list property keys.

Save the changes. To see them, open a new Query Window.

5. Configure the property keys for Movie form windows.

   In the Query Window, click New to create a Movie form window. The Assistant updates to set the Task pull-down list to form and the Entity pull-down list to Movie.

   Examine the form window. As is most commonly the case, the default property keys for the form task are correct for this application. In general, the form window should allow users to edit all of an entity's client class properties. However, you might want to reorder the properties so they appear in different locations in the form window.

   Using the up and down buttons, order the properties as follows: **title**, **category**, **dateReleased**, **revenue** (move up), **posterName**, **trailerName**, **plotSummary**, **directors**, **roles**, **voting**, **reviews**, **studio**. Save the changes, and open a new Movie form window to see how they affect the form window layout.

Configuring Widgets

The Assistant's Widgets tab allows you to configure the widgets used to display and manipulate properties (attributes and relationships). For example, you can change the widget that's used to display a particular attribute, and you can set the widget's resizing behavior.

1. Change the widget for the PlotSummary entity's **summary** attribute to a text area.

   Select the Widget tab in the Assistant. Set the Task pull-down list to form, the Entity pull-down list to PlotSummary, and the Property pull-down list to **summary**.

Warning: Don't use Movie as the Entity and **plotSummary** as the property key. You're configuring the widget used to display the **summary** property of the PlotSummary entity, not the widget for displaying the **plotSummary** relationship of the Movie entity. So be sure the Entity pull-down list is set to PlotSummary and the Property pull-down list is set to **summary**.

Set the Widget Type to EOTextAreaController. Save the change, and open a new Movie form window to see the change.

2. Remove the Summary label.

   The **summary** Widget label is redundant with the PlotSummary tab label, so you should remove it.

   If you opened a new Movie form window to see the change you made in step 1, you first need to restore the Assistant's Widget tab settings. Set the Task to form, the Entity to PlotSummary, and the Property to **summary**.

   Now set the **summary** property's "Show Label Component" setting to False. Save the change, and open a new form to see it.

3. Set sizing information for the **summary** widget.

   Resize the window so it's much larger than the default size. Notice that the **summary** widget doesn't resize vertically. To prevent the waste of screen real estate and to display more of the summary upon window resizing, you need to set the widget to resize.

   Restore the Assistant Task, Entity, and Property settings to form, PlotSummary, and **summary**, respectively. Now set the **summary** property's "Vertically Resizable" setting to True. Save the change, and open a new form to see it.

4. Change the widget for the Review entity's **review** attribute to a text area.

   In the Widget tab of the Assistant, set the Task to form, Entity to Review, and Property to **review**. Set the Widget Type to EOTextAreaController. Save the change.

5. Set sizing information for the **review** widget.

In a new Movie form window, switch to the Review tab. Resize the window so it's much larger than the default size. Notice that the Review widget doesn't resize vertically. In this window, this behavior is appropriate because the table view above the Review widget does resize to take up the added area.

However, you might want a larger area in which to display movie reviews. If you set the text area to resize vertically, the widget still won't resize because the table view above it does. So to provide a larger text area, you can set a minimum height.

Restore the Assistant Task, Entity, and Property settings to form, Review, and **review**, respectively. Now set the **review** property's "Minimum Height" setting to 200. Save the change, and open a new form to see it.

Typical Workflow

The most natural way to customize an application's property keys and widgets is to examine and modify one window at a time, as outlined by the steps below:

1. With the Assistant open, click the first tab in the Query Window.
2. Examine the search properties for the current entity. Remove any fields necessary, add others, and then order the property keys as you want them to appear in the Query Window.
3. Choose list in the Assistant's Task pull-down list, and modify the list property keys. Since the Query Window displays list properties as well as query properties, you can see what changes you need to make.
4. Click the next tab in the Query Window. The Assistant automatically updates to track your movements in the application. This saves you setting the Entity pull-down list in the Assistant.
5. Repeat steps 2 through 4, until you've set the query and list property keys for all the main entities.
6. Click the first tab in the Query Window.
7. Click New to open a form window for the current entity. Make necessary changes to the entity's form property keys.

8. Make any necessary changes to property keys of any other entity that appears in the current entity's form window. To do so, choose list or identify in the Assistant's Task pull-down list, and set the Entity pull-down list to the other entity.

9. Click the next tab in the Query Window.

10. Repeat steps 7 and 9.

The following sections demonstrate this approach. Before continuing, open the Assistant if it isn't open already, and select the Properties tab.

Examining the Panes of the Query Window

You've already configured the Movie pane of the Query Window, so examine the next pane, Talent. The window allows you to search on a talent's first and last names, and it shows the first and last names in the results table. This is fine, so move on to the next pane, Studio.

Configuring the Studio Pane of the Query Window

The Studio pane allows you to search for studios by their name and budget. It seems unlikely that users want to search on budget.

1. Remove **budget** from the query property keys list.

The results table shows the studio name and budget. This seems fine, so move on to the next pane.

Configuring the Customer Pane of the Query Window

The Customer pane allows you to search for customers by their city, first name, last name, credit card number, and credit card authorization. Users should not look up customers by credit card information, so you should remove it.

1. Remove **creditCard** from the query property keys.

   The **creditCard** property is a to-one relationship that Direct to Java Client expands to the identify property keys (**cardNumber** and **authorizationNum**) of the relationship's destination entity (a CreditCard). By removing **creditCard** from the property keys, you remove the **cardNumber** and **authorizationNum** fields from the query interface.

2. Add **memberSince** and **phone** to the property keys.

   Two other Customer properties, **memberSince** and **phone**, seem like reasonable attributes to query on, so add them.

3. Order the keys as follows: **firstName**, **lastName**, **city**, **phone**, **memberSince**.

4. Modify the list property keys.

   The results table is far too crowded with properties to be useful, so pare down the set of list property keys. Remove **state**, **streetAddress**, **zip**, and **creditCard** from the list property keys. Order the remaining keys as follows: **firstName**, **lastName**, **city**, **phone**, **memberSince**.

Move on to the next pane.

Configuring the Unit Pane of the Query Window

The Unit pane allows you to search for units by unit ID, date acquired, movie name, and movie release date. It seems unlikely that users will want to search for particular units by movie release date, so you'll remove it.

Note that the query property keys for Unit are **unitID**, **dateAcquired**, and **video**. Through the **video** key, Direct to Java Client gets both a movie name and movie release date (which correspond to the Movie identify property keys). You want to keep the movie name in the interface, but not the movie release date.

1. Remove **video** from the property keys.

   Using the **video** key, you can't get one identify Movie property without the other one, so remove it. You'll add the Movie **name** property to the query interface another way.

2. Add **video.movie.title** to the property keys.

   At the bottom of the Assistant Properties pane, there's a field labeled "Additional Property Key Path". In this field, type `video.movie.title`, and click Add.

3. Save and restart to observe these changes.

   Adding key paths to an entity's property keys flattens attributes into the entity for display purposes: the attribute identified by the key path is

treated in the user interface as if it's one of the entity's own attributes rather than an attribute of a related entity.

4. Change the label for **video.movie.title**.

   The label for **video.movie.title**, "Video Movie Title", seems unnecessarily wordy. "Movie Title" would be better.

   Select the Widgets tab in the Assistant. Set the Task pull-down list to <ALL> if it isn't already, then select **video.movie.title** in the Property Key pull-down list. In the Label field, delete the word "Video" so the label is simply "Movie Title".

5. Save and restart to observe this change.

   Notice that the movie title label is now "Movie Title" for both the search field and for the column in the results table. To set one, but not the other, you would simply set the Task pull-down list to the appropriate task before making the change.

6. Examine Unit's list properties.

   The **notes** property contains free-form text, and doesn't display well in a list, so remove **notes** from Unit's list property keys.

Move on to the next pane.

Configuring the Video Pane of the Query Window

Video's query interface has the same problem as Unit's. It seems unlikely that users will want to search for videos by movie release date. Remove it from the query interface the same way you did for Unit. That is, remove the **movie** relationship from the property keys and add **movie.title**.

Examining Form Windows

Now that you've configured the panes of the Query Window, move on to the form windows. You've already configured Movie's form window, so start with Talent.

Open a Talent form. It looks fine, so move on to Studio. It, too, looks fine, so move on to Customer.

Configuring the Customer Form Window

All the Customer properties are property keys. This is fine; users need to be able to edit all of a customer's data. However, the properties are ordered poorly. A customer's name should come first, and the components of a customer's address should be grouped together and ordered so they appear as a proper address.

1. Order the Customer form property keys as follows: **firstName**, **lastName**, **memberSince**, **phone**, **streetAddress**, **city**, **state**, **zip**, **creditCard**, **rentals**.

2. Examine the Credit Card pane at the bottom of the Customer window.

   The property keys are poorly ordered, and the "Num" in the "Authorization Num" label should be spelled out.

3. Using the Properties Assistant, order CreditCard's form property keys as: **cardNumber**, **expirationDate**, **limit**, **authorizationNum**, **authorizationDate**, **customer**.

   Note that **customer** is a to-one relationship from CreditCard to Customer. Ordinarily, a to-one relationship is represented in a form, but **customer** isn't represented here. That's because Direct to Java Client filters the **customer** property key out of the user interface rather than display it recursively. Since the CreditCard form is contained within a Customer form, Customer information is already displayed. This feature is called *entity hierarchy filtering*.

Warning: Be careful about removing relationships from the set of property keys. Even though a relationship isn't necessary for a window you're looking at, it might be needed by another window.

As an example, consider the MovieRole entity, which has to-one relationships to Movie and Talent. Similarly, Movie and Talent have to-many relationships back to their MovieRoles. Since MovieRole is an "other" entity, MovieRole records are manipulated in a master-detail user interface in Movie and Talent form windows. In a Movie form window, the interface for displaying and setting a MovieRole's value uses only the **roleName** attribute and the **talent** relationship. Entity hierarchy filtering prevents the interface from using the **movie** relationship.

When you're looking at the Movie form window, you might consider removing MovieRole's **movie** relationship from the form task's property keys. It's filtered out, but you might think of removing it anyway for clarity. However, if you remove it, the **movie** relationship won't be displayed in the Talent form window either. In the Talent window a MovieRole's **movie** relationship is important information.

4. In the Widgets pane of the Assistant, change the **authorizationNum** label to "Authorization Number."

5. Examine the Rentals pane.

   The Rentals pane looks OK, so move on to the next form window.

Configuring the Unit Form Window

The form property keys for Unit are correct, and their order is fine, too. However, the notes property should be in a text area instead of a field.

1. Change the **notes** widget to a text area.

   Select the Widgets tab in the Assistant. Set the Task pull-down list to form, and select **notes** in the Property Key pull-down list. Set the Widget Type to EOTextAreaController. Save the change, and open a new Movie form window to see the change.

2. Examine the Rental area of the window.

You need to resize the windows to read the Rental table columns. Each Rental line displays Customer identify properties: **city**, **firstName**, and **lastName**. The **city** attribute isn't really necessary, and it would be more helpful if a customer's last name were to appear first.

3. Modify Customer's identify property keys.

   Remove **city** from Customer's identify property keys. Order **lastName** first. Save your changes.

Move on to the next form window, Video. It's fine.

Configuring Windows

After configuring properties and widgets, the bulk of your Assistant customizations are done. The final change you'll make to your application in this tutorial is to change the title of a window. To do this, you use the Windows tab of the Assistant.

The Windows tab allows you to do things such as the following:

- Set the title of windows.
- Change the default position and size of a window.
- Specify whether to save window position and size in user defaults.
- Specify disposal and reuse behavior to tune performance.

Changing the Title of the Query Window

To change the title of the Query Window to "Find", perform the following steps.

1. Select the Windows tab in the Assistant.
2. Click the Query Window.

   This focuses the Assistant on the Query Window, setting the Question to window, setting Task to queryWindow, and disabling Entity (because the Query Window displays multiple entities).

3. In the Label field, type "Find".
4. Save your changes and restart the application.

The Query Window's title is now "Find."

Other Assistant Settings

In addition to Entities, Properties, Widgets, and Windows tabs, the Assistant also provides Miscellaneous and XML tabs. The Miscellaneous tab allows you to specify application wide settings for alignment and label placement (for all widgets in all windows for all entities).

The XML tab is more advanced. It allows you to view the output of the rules system. The output, which is the resolution of rules, is represented as XML. As you become more familiar with the Direct to Java Client technology, you can read the XML to verify user interface configurations instead of restarting the application to see the actual effects of your changes.

You can also use the XML tab to save the XML to a file. You typically do this when you want to customize an application by *freezing* it's XML. Freezing allows you to explicitly state the result of a rule system request. The freezing technique is beyond the scope of this tutorial.

Advantages of Direct to Java Client

Direct to Java Client dynamically generates the user interfaces for Java Client applications or for parts of them. The advantage of this technology is that it's not necessary to write source code for controlling the user interface. As a consequence, Direct to Java Client:

- Flattens the learning curve for developing applications
- Reduces the time required to develop applications
- Reduces the number of causes of errors
- Increases the maintainability and adaptability of applications
- Increases prototyping capabilities
- Allows you to focus on business logic instead of on the user interface

Also, Direct to Java Client applications are constructed using well-tested Apple technology, which increases the stability of applications and reduces the time required to test applications before deploying them.

Chapter 2

Understanding Direct to Java Client Applications

Direct to Java Client applications are very simple to create because so much of your application simply works out of the box. You don't write the code, but the functionality is nevertheless in the application. It's provided by numerous Java packages that are part of WebObjects.

Even though you don't write the code, you can customize it's behavior. This chapter explains how Direct to Java Client applications work, so you can understand the many different ways you can tailor them to your needs.

Architectural Overview

Direct to Java Client applications are three tier, comprising a database server, a server application, and a client application. Figure 1 shows the three tiers and the Java packages used in them.

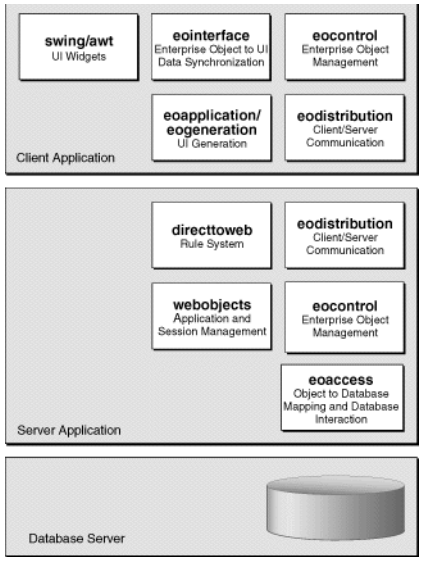

Figure 1. The Components of a Direct to Java Client Application

All of the packages except eoapplication, eogeneration, and directtoweb are used in exactly the same way in a Direct to Java Client application as they are in a Java Client application (an application for which you provide the user interface):

- *The webobjects package.* Used in the server application to provide application and session management.
- *The eocontrol package.* Used in both the server and client applications to manage a graph of enterprise objects (object representations of database records). It translates operations performed on enterprise objects into database operations, thereby keeping the object graph and the database in sync.
- *The eoaccess package.* Used in the server application to handle interactions with the database, using a model file to map database records to enterprise objects.
- *The eodistribution package.* Used in both the server and client applications to handle communication between the server and application processes.
- *The swing and awt packages.* Used in the client application to implement the user interface.
- *The eointerface package.* Used in the client application to synchronize the user interface with enterprise objects. The eointerface package populates user interface widgets with data from enterprise objects. When a user makes changes to the data in the user interface, eointerface notices the changes and make the corresponding changes to the appropriate enterprise objects.

The remaining packages distinguish a regular Java Client application from a Direct to Java Client application.

- *The eoapplication and eogeneration packages.* Used in the client application to dynamically generate the user interface. Controller classes in these packages define application-level functionality. A hierarchy of these controllers generates and manages an application's user interface. Note that eoapplication contains classes that can be used outside of a Direct to Java Client application. For example, eoapplication's EOApplication object is generally used in regular Java Client applications, as well.
- *The directtoweb package.* Provides a server side rule system that the client side eoapplication and eogeneration packages use to generate the user interface. The rules specify how to configure the objects in the eoapplication and eogeneration controller hierarchy for a particular Direct to Java Client application.

The way these packages provide their functionality is discussed in more detail in the following sections.

Controller Hierarchy

The most interesting part of the client side portion of a Direct to Java Client application is the *controller hierarchy*. The controller objects in the hierarchy are responsible for generating and managing the client application's user interface. The hierarchy as a whole describes the complete functionality of an application.

The controller hierarchy mirrors the hierarchy of windows and widgets that make up the client application's user interface. The root of the hierarchy is an EOApplication object. The EOApplication's subcontrollers are usually window or applet controllers, which themselves have subcontrollers.

For example, consider the select Studio dialog from the Tutorial application (Figure 2).

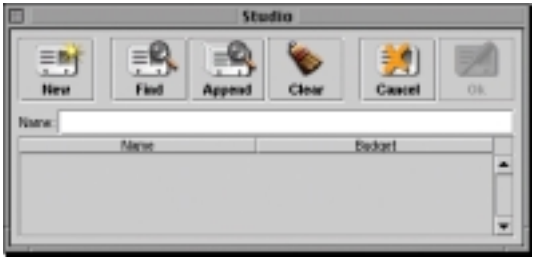

Figure 2. Select Studio Dialog

The branch of the controller hierarchy for this dialog looks like this:

```
EOModalDialogController
    EOActionButtonsController
        EOQueryController
           EOTextFieldController
           EOListController
               EOTableController
                   EOTableColumnController
                   EOTableColumnController
```

The EOModalDialogController manages the select Studio dialog itself, the EOActionButtonsController manages the row of buttons at the top of the dialog, and so on.

Controllers

The objects in the controller hierarchy are instances of EOController subclasses. The EOController class defines basic controller behavior. The most significant functionality is managing the controller hierarchy (building, connecting, and traversing the hierarchy) and handling actions. Controllers define actions that users can perform (such as clicking a Find button) and they know how to respond to those actions when they're performed.

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 (usually an EODynamicApplication object) is the root of an application's controller hierarchy.
- User interface level controllers manage portions of an application's user interfaces, such as windows and tab views. They determine the layout of their subcontrollers, resizing behavior, and so on. In the controller hierarchy for the select Studio dialog, the EOModalDialogController and the EOActionButtonsController are user interface level controllers.
- 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. The entity level controllers in the select Studio dialog hierarchy are the EOQueryController and the EOListController.
- Property level controllers manage widgets for displaying properties. In the select Studio dialog hierarchy, the EOTextFieldController and the EOTableColumnControllers are property level controllers. Their function in this dialog is to provide the widgets for entering (EOTextFieldController) and displaying (EOTableColumnControllers) Studio properties.

Creating the Controller Hierarchy

The process for creating the controller hierarchy involves an EOControllerFactory object, a rule system, and D2WComponents.

An EOControllerFactory in the client application is created during a client application's initialization, and the controller factory in turn creates the controller hierarchy. To do so, it uses a server side rule system, which provides XML descriptions of controller hierarchies. The controller factory, using an EOXMLUnarchiver object, parses the XML and generates the specified controllers.

Unarchiving XML

To see how an EOXMLUnarchiver processes the XML returned from the server, consider the following XML:

```
<MODALDIALOGCONTROLLER reuseMode="ReuseIfInvisible"
disposeIfDeactivated="false" typeName="question = modalDialog,
task = select, entity = Studio">
    <ACTIONBUTTONSCONTROLLER widgetPosition="Top">
        <QUERYCONTROLLER entity="Studio" minimumWidth="256"
        alignsComponents="true">
            <TEXTFIELDCONTROLLER valueKey="name"
            isQueryWidget="true"/>
            <LISTCONTROLLER minimumWidth="256" editability="Never"
            alignsComponents="true" entity="Studio">
                <TABLECONTROLLER>
                    <TABLECOLUMNCONTROLLER editability="Never"
                    valueKey="name"/>
                    <TABLECOLUMNCONTROLLER valueKey="budget"
                    formatPattern="#,##0.00;-#,##0.00"
                    editability="Never"
                    formatClass=
                    "com.apple.client.foundation.NSNumberFormatter"/>
                </TABLECONTROLLER>
            </LISTCONTROLLER>
        </QUERYCONTROLLER>
    </ACTIONBUTTONSCONTROLLER>
</MODALDIALOGCONTROLLER>
```

This XML describes the controller hierarchy for the Tutorial application's select Studio dialog. The XML tags—**MODALDIALOGCONTROLLER**, **ACTIONBUTTONSCONTROLLER**, and so on— specify a controller class. The XML attributes—**reuseMode**, **disposeIfDeactivated**, and so on—specify how to configure the corresponding controller.

The EOXMLUnarchiver maps XML tags to particular EOController classes. For example, the following table describes the mappings for the tags in the select Studio dialog XML.

XML Tag	Controller Class
`MODALDIALOGCONTROLLER`	EOModalDialogController
`ACTIONBUTTONSCONTROLLER`	EOActionButtonsController
`QUERYCONTROLLER`	EOQueryController
`TEXTFIELDCONTROLLER`	EOTextFieldController
`LISTCONTROLLER`	EOListController
`TABLECONTROLLER`	EOTableController
`TABLECOLUMNCONTROLLER`	EOTableColumnController

Note: You can change the controller class with which a particular XML tag corresponds by writing a custom rule or by freezing XML. Writing custom rules and freezing XML are discussed in the chapter "Customizing Direct to Java Client Applications" (page 55).

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

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

These attributes correspond to the EOTextField methods **setValueKey** and **setIsQueryWidget**. The `valueKey="name"` specifies that the text field controller corresponds to a property named "name" (in this case, the **name** attribute of a Studio object). The `isQueryWidget="true"` specifies that the text field is used to get search criteria from the user and not to display and edit a property's value

For information on the XML tags and attributes for the EOController classes, see the *EOApplication Framework Reference* and the *EOGeneration Framework Reference*.

Server Side XML Generation

When the controller factory sends a request to the server application, how does the XML get generated on the server side? The work is split between the rule system and a set of D2WComponent classes.

The rule system receives the controller factory's requests. It evaluates rules to see what D2WComponent subclass should generate the XML for the current request, and then that component does the actual generation.

The Rule System

The rule system is a private system that's provided by the **com.apple.yellow.directtoweb** package (it's the same rule system as the one used by Direct to Web). You never need to access it directly; the EOControllerFactory takes care of all interactions with the rule system. Nevertheless, it's important to understand the basics of the rule system and of how the controller factory interacts with it.

All the information about how to configure a Direct to Java Client application is stored in the form of *rules*. A rule has a key, a condition that must be true for the rule to "fire", a value, and a priority. The rule system evaluates requests as follows:

1. The controller factory makes a request to the rule system by specifying a key.

2. The rule system identifies the rules whose key is the same as the request key.

3. It then evaluates the conditions of the matching rules to see which can fire.

4. Of the rules that can fire, the rule system fires the one with the highest priority, returning the value for the rule's key.

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 rules' conditions. 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 = queryWindow
- entity = Movie, question = window, task = form
- entity = Studio, question = modalDialog, task = select

A specification always contains a "question" entry, which contains the request key. The request keys in the above specifications are "window", "window", and "modalDialog", respectively.

The rule system stores the key-value pairs of state information in a D2WContext object. The D2WContext's whole purpose is to keep track of state as a page is generated. Initially the D2WContext is filled with state information provided by the controller factory. As the rule system processes the request, however, the system adds more and more state.

D2WComponents

There's a one-to-one correspondence between the Direct to Java Client D2WComponent subclasses and the client side EOControllers. For example, on the client side, there's an EOTextFieldController. The corresponding server side D2WComponent class is also named EOTextFieldController. The client side class creates and manages user interface widgets, while the server side class generates XML to describe how to configure the client side controller.

The Direct to Java Client D2WComponents have **.wo**s with **.html** and **.wod** files, which you can see in the DirectToJavaClient framework.

1. Navigate to the DirectToJavaClient framework (in **/System/Library/Frameworks** on Mac OS X Server or in **$NEXT_ROOT/Library/Frameworks** on Windows).

2. Navigate to the Resources folder.

   In the Resources folder you can see the **.wo** files for all the Direct to Java Client D2WComponents.

3. Double-click the **EOBoxController.wo** to open it in WebObjects Builder.

The **EOBoxController.html** contains XML instead of the typical HTML, so WebObjects Builder must open the component in source view mode.

Rule System Requests

The user interface of a Direct to Java Client application isn't generated all at once. Instead it's generated piecemeal as each new window is activated. Correspondingly, the controller factory makes rule system requests as each new window is activated.

When an application starts up, the controller factory makes requests for the following keys:

- **availableSpecifications** which tells the controller factory all the specifications (dictionaries of request keys and state information) that are valid for the application. A Direct to Java Client application caches this information to avoid unnecessary round trips to the server later on as it generates windows.
- **defaultSpecifications** which tells the controller factory which windows to open automatically once the application has finished initializing.
- **actions** which tells the controller factory what actions to add to the main menu along with standard menu items such as Quit and Edit. For example, you could add an action to open a form or query window.

Then, to generate the controller hierarchy for a window or dialog, the controller factory makes requests for the following keys:

- **window** which returns the controller hierarchy XML for the window the application will open next. In the request's specification, the controller factory must also provide state information (typically a task and optionally an entity) so the rule system can determine what window is being generated.
- **modalDialog** which returns the controller hierarchy XML for the dialog the application will open next. In the request's specification, the controller factory provides a task and optionally an entity.

As the rule system generates the controller hierarchy XML, it can make additional rule system requests. For example, to generate the XML for a select

Studio dialog, the rule system must know what the Studio entity's query and list properties are. This information is also stored in the form of rules.

Internal Rule System Requests

When the rule system evaluates a request from the controller factory, the actual returned value is 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 the XML, the D2WController object might require the rule system to evaluate additional requests. The most significant of which are:

- **controller** which identifies an entity level controller for a task and entity identified in the request's specification.The entity level controller defines a part of a window or dialog's user interface for performing the specified task on the specified entity. For example, there's an EOListController that uses a table view to display a list of a particular entity's objects.
- **propertyKeys** which identifies the property keys for a task and 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.

Generating the Select Studio Dialog

As an example of how the Direct to Java Client D2WComponent classes work, consider the select Studio dialog. Suppose a user clicks the Select button on a Movie form window to select a new Studio for the Movie. The controller factory then makes a request to the rule system with the following specification:

entity = Studio, question = modalDialog, task = select

The default rule fired to satisfy this request is:

- Condition (Lhs): *true* (No condition is specified, so condition is always true.)
- Key (Rhs key): modalDialog
- Value (Rhs value): EOModalDialog
- Priority: 0

Thus, the D2WComponent that generates the XML for the select Studio dialog is EOModalDialog. The EOModalDialog's **.wo** contains:

EOModalDialog.html (XML)

```
<WEBOBJECT name=modalDialogController>
    <WEBOBJECT name=actionWidgetController>
        <WEBOBJECT name=taskController/>
    </WEBOBJECT>
    <WEBOBJECT name=content/>
</WEBOBJECT>
```

EOModalDialog.wod

```
modalDialogController: EOSwitchComponent {
    componentNameKey = "modalDialogController";
    d2wContext = localContext;
    controllerType = "modalDialogController";
}
actionWidgetController: EOSwitchComponent {
    componentNameKey = "actionWidgetController";
    d2wContext = localContext;
    controllerType = "actionWidgetController";
    forceHorizontallyNotResizable = noValue;
    forceVerticallyNotResizable = noValue;
    isRootController = "false";
}
taskController: EOSwitchComponent {
    componentNameKey = "controller";
    d2wContext = localContext;
    controllerType = noValue;
    forceHorizontallyNotResizable = noValue;
    forceVerticallyNotResizable = noValue;
    isRootController = "false";
}
content: WOComponentContent {
}
```

The EOSwitchComponent you see in the **.wod** file is a private dynamic element that makes a new rule system request using the **componentNameKey** as the request key. So for example, consider the **modalDialogController**. The switch component makes a new rule system request with the key "modalDialogController" (the value 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. The remaining bindings (any binding other than **componentNameKey** and **d2wContext**) identify additional state that the switch component adds to the new D2WContext. For the **modalDialogController**, the additional state is simply that the **controllerType** is "modalDialogController".

In this manner, the XML hierarchy is built recursively using switch components. One of the leaf nodes in the select Studio dialog is for an EOTextFieldController who's **.wod** file looks like this:

```
controller: WOXMLNode {
    elementName = "TEXTFIELDCONTROLLER";
    alignment = d2wContext.alignment;
    alignmentWidth = d2wContext.alignmentWidth;
    alignsComponents = d2wContext.alignsComponents;
    className = d2wContext.className;
    displayGroupProviderMethodName =
        d2wContext.displayGroupProviderMethodName;
    editability = d2wContext.editability;
    enabledDisplayGroupProviderMethodName =
        d2wContext.enabledDisplayGroupProviderMethodName;
    enabledKey = d2wContext.enabledKey;
    formatAllowed = d2wContext.formatAllowed;
    formatClass = d2wContext.formatClass;
    formatPattern = d2wContext.formatPattern;
    highlight = d2wContext.highlight;
    horizontallyResizable = d2wContext.horizontallyResizable;
    iconName = d2wContext.iconName;
    iconURL = d2wContext.iconURL;
    isQueryWidget = d2wContext.isQueryWidget;
    label = d2wContext.label;
    labelAlignment = d2wContext.labelAlignment;
    labelComponentPosition = d2wContext.labelComponentPosition;
    minimumHeight = d2wContext.minimumHeight;
    minimumWidth = d2wContext.minimumWidth;
    prefersIconOnly = d2wContext.prefersIconOnly;
    usesHorizontalLayout = d2wContext.usesHorizontalLayout;
    usesLabelComponent = d2wContext.usesLabelComponent;
    valueKey = d2wContext.propertyKey;
    verticallyResizable = d2wContext.verticallyResizable;
}
content: WOComponentContent {
}
```

A WOXMLNode is a component that generates XML for a node in the controller hierarchy. It's bindings tell the server side D2WComponent how to configure it's client side counterpart. For example, the binding names in the EOTextFieldController **.wod** file correspond to XML attributes understood by the client side EOTextFieldController. 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.

For example, the client side EOTextFieldController uses the XML attribute **usesLabelComponent** to specify whether it should generate a label for its text field widget. The server side EOTextFieldController assigns a value to the **usesLabelComponent** XML attribute based on state stored in the rule system's D2WContext.

Chapter 3

Customizing Direct to Java Client Applications

The default application generated by Direct to Java Client doesn't usually meet all of an application's requirements, so you generally need to customize an application.

There are many approaches to customizing a Direct to Java Client application. This chapter discusses each, describing how to use them, what they're good for, their maintenance costs, and their other advantages and disadvantages.

Customization Approaches

The following list describes the most common approaches you can take to customize a Direct to Java Client application (from easiest and most maintainable to most advanced and least flexible):

- Using the Assistant

  This is the easiest way to customize a Direct to Java Client application. The Direct to Web Assistant is an easy-to-use tool that is integrated into a running client application. It allows you to perform the most common customizations, directly test them while the application is running, and save them in your project. To see the kind of customizations you can do with the Assistant, work through "Getting Started with Direct to Java Client" (page 11).

  Use this technique to categorize entities as main, enumeration, or other; to customize the query, list, form, and identify properties; to customize windows, dialogs, and other widgets.

- Writing custom rules

  All the information about how to configure a Direct to Java Client application is stored in the form of rules. The default rules generate the default Direct to Java Client application. Adding new rules that override or supplement the default rules is an easy to maintain approach that doesn't interfere with your use of the Assistant.

  This technique is useful to configure many different controllers with a single simple rule. For more information on when and how to write custom rules, see "Writing Custom Rules" (page 59).

- Freezing XML

  Freezing XML allows you to explicitly state the result of a rule system request. In other words, instead of using the rule system to provide XML, the controller factory uses XML that you supply to generate the controller hierarchy.

  This approach is less flexible than the previous techniques. Changes to the set of rules and to an application's models don't have an effect on the controller hierarchy for the frozen XML. You have to edit the XML by hand to keep it up to date with other project changes.

  Nevertheless, it's sometimes useful to freeze XML. For example, it can be difficult to express a particular aspect of a controller hierarchy using rules. Also, it's sometimes easier to understand how to modify frozen XML than it is to write new rules or source code.

  You can also consider freezing XML for problematic windows whose XML generation takes too long. If you do this, be sure to wait until the end of the development process so you postpone the maintenance issues as long as possible.

  For more information on this technique, see "Freezing XML" (page 63).

- Freezing nib files

  Freezing nib files allows you to completely specify user interface layout by providing a nib file. This approach has many disadvantages: maintenance is more difficult, platform specific layout is harder to achieve, EOModel changes can't be picked up as easily, and so on. Consequently, freezing nib files should be used very carefully.

  Nevertheless, it's sometimes useful to specify portions of the user interface with frozen nib files. For example, non-engineers such as graphic and user interface designers can modify nib files directly to get fine grained control over the interface.

  For more information on this approach, see "Freezing Nib Files" (page 68).

Other Approaches

There are also more specialized ways to change the way Direct to Java Client works. You use the following techniques to accomplish specific tasks:

- Adding custom actions

  Direct to Java Client provides hooks you can use to introduce custom commands into an application's main menu. For more information, see "Adding Custom Actions" (page 69).

- Implementing custom controller classes

  This approach allows you to change the way an application performs a particular task or to add new functionality to the default set. For more information, see "Implementing Custom Controller Classes" (page 73).

- Implementing controller factory delegate methods

  The EOControllerFactory allows you to modify many aspects of the default behavior by providing a delegate. For example, you can use this technique to alter the specifications the controller factory supplies to the rule system, substitute a controller for the one returned by the rule system, change the property keys used for a particular entity and task, and so on.

  If you create a controller factory delegate, be careful to avoid implementing unsecure functionality in the controller factory. Information that isn't usually visible on the client should not be sent to the client only to use it in the factory's delegate.

Writing Custom Rules

There are two components of Direct to Java Client that determine the user interface of an application: controllers and rules. Controllers determine user interfaces for performing specific operations, and rules determine how to configure the controllers.

The influence rules have on an application range from very significant to very subtle. For example, a rule can specify what controller to use, substituting a custom controller that introduces a completely new user interface for a task. Or a rule can merely specify parameters that control a controller's behavior in small ways, such as setting a minimum size for a widget.

The easiest way to customize a Direct to Java Client application is to add new rules. This is how the Assistant works. As you configure an application with the Assistant, it's creating rules to achieve the effects you want. The priority of the rules created by the Assistant is higher than the priority of the default rules. So when you configure an application with the Assistant, the corresponding Assistant-created rules fire instead of the default rules.

If the Assistant can't perform the customizations you want, you can create rules by hand with an application called the Rule Editor. When you create your own rules, you specify the priority. Therefore, you determine whether your custom rules take precedence over other rules.

Using the Rule Editor

Suppose you want to right align all text fields that contain numeric data. The Assistant isn't sophisticated enough to allow you to do this. However, you can configure your application to have this alignment behavior by writing a custom rule. The procedure is as follows:

1. Start the Rule Editor.

   On Mac OS X Server, choose WebObjects ▸ Rule Editor in the Apple menu.

   On Windows, choose Rule Editor in the WebObjects program group.

   When the Rule Editor starts, it opens a new, untitled rule file.

2. Click New to create a new rule.

3. Set the rule's condition, key, and value.

   Note that the Rule Editor uses formal rule terminology to refer to parts of a rule. Formally, a rule is expressed as follows:

   ```
   (condition) => key = value
   ```

   The condition of a rule is known as the left hand side of the rule (lhs), the key is known as the right hand side key (rhs key), and the value is known as the right hand side value (rhs value). For example, a rule that right aligns numeric text fields can be expressed as follows:

```
(not (attribute = nil)) and ((attribute.valueClassName =
'NSNumber') or (attribute.valueClassName = 'NSDecimalNumber'))
=> alignment = “Right”
```

In this rule, the condition or lhs is:

```
((not (attribute = nil)) and ((attribute.valueClassName =
'NSNumber') or (attribute.valueClassName = 'NSDecimalNumber')))
```

The key or rhs key is “alignment”, and the value or rhs value is “Right.”

4. Set the rule’s priority to 50.

   The priority of the Assistant’s rules is 100. Use a lower priority than 100 if you want the Assistant’s rules to be preferred to your own rules, or a higher priority otherwise. Don’t create rules that have the same priority as the Assistant’s rules, because this can confuse the Assistant.

   The priority of the default rules is 0, so both the Assistant’s and your custom rules are preferred to the default rules.

5. Set the rule’s assignment class to Assignment.

   In the Class pop-up list on the right side of the window, select Assignment. This class is selected by default, so you shouldn’t actually have to change it. Other assignment classes allow you to programmatically determine rule results. For more information, see *Developing WebObjects Applications with Direct to Web*.

6. Name the rule file **d2w.d2wmodel** and save it in the Resources folder of your application’s project.

   Rules are stored in files with the extension **.d2wmodel**.

   In addition to your custom rule file, there are two other rule files that are used by an application: a file containing the default rules and a file containing the Assistant’s rules. The default rules are also in a file named **d2w.d2wmodel**, but the default rule file is located in the Resources folder of the DirectToJavaClient framework. The other rule file is the Assistant’s rule file. It is located in the Resources folder of an application’s project along with the custom rules file, but it has the name **user.d2wmodel**.

   You can look at the Assistant rules by opening the **user.d2wmodel**; simply double-click the file, which opens it in the Rule Editor. However, don't

edit the Assistant's file because the Assistant writes out the file whenever it saves, removing rules it doesn't create itself. By storing your custom rules in the **d2w.d2wmodel** file, you can write custom rules and still use the Assistant.

When to Write Custom Rules

Custom rules are used for a variety of tasks, including the following:

- Freezing XML (see "Freezing XML" on page 63)
- Freezing nibs (see "Freezing Nib Files" on page 68)
- Introducing custom controllers (see "Implementing Custom Controller Classes" on page 73)

However, the most common reason to write custom rules is to configure many different controllers with one simple rule. Right justifying numeric text fields as described earlier in this section is a good example. It's possible to use the Assistant to accomplish the same thing as the custom rule, but using the Assistant has a drawback in this scenario: Each text field has to be configured individually, which is time consuming and error prone. Not only is it easy to overlook a text field; but if the model changes in the future, you have to remember to configure any newly added numeric text fields.

As another example, suppose you want to set the minimum width of all an application's windows. Again this is something you can do in the Assistant, but you have to do it individually for each window. The following single rule can accomplish the same thing:

- Condition (Lhs): controllerType = 'windowController'
- Request key (Rhs key): minimumWidth
- Value (Rhs value): 512
- Priority: 50

In this rule, it's necessary to specify what kind of controller the rule system is working on. Many different controllers have the attribute **minimumWidth**, so to limit this rule's scope to setting the minimum width of only windows, you must restrict the rule to firing only when the rule system is working on window controllers.

The rule system's D2WContext stores the type of controller it's working on and makes the type available with the key **controllerType**. The possible values for **controllerType** are:

- windowController
- modalDialogController
- entityController
- widgetController
- tableController
- groupingController
- dividingController
- actionWidgetController

Note: You can use any state information stored in the D2WContext in the condition (lhs) of a rule. For more information on the entries in a D2WContext, see the section "Server Side XML Generation" (page 47).

Trouble Shooting Custom Rules

- If your custom rules don't seem to be firing...

  Restart the server application and try again. The rule system caches the results of its rules, so you have to restart the server application before changes to a rule file are in effect.

- If rule results aren't consistent...

  Ensure that the significant keys of the rule model are complete. For more information, see *Developing WebObjects Applications with Direct to Web*.

Freezing XML

Recall that the client side controller factory generates the user interface for a window at a time. To create a window, the controller factory makes a rule system request, specifying the window's task (form, query, or select), the window type (window or modal dialog), and the entity (if required by the task). The rule system fires a rule that outputs a D2WComponent, which in turn provides XML that describes the controller hierarchy for the current window.

By default, the rule system returns a D2WComponent that generates the XML dynamically. However, you can create a custom D2WComponent that returns static or *frozen* XML. This is called "freezing XML".

The basic steps for freezing XML for a particular window are:

1. Create a D2WComponent to supply the frozen XML.
2. Use the Assistant to dynamically generate the XML and then save it to a file.
3. Copy the frozen XML and paste it into your D2WComponent's **.html** file.
4. Modify the XML file to specify the customizations you want.
5. Write a rule that uses your D2WComponent to supply the frozen XML.

The most common reason to freeze XML is to specify a custom user interface layout that is difficult or impossible to express with rules. For example, suppose that you want to modify the select Studio dialog to put the **name** query text field in a box. This is very easy to specify using frozen XML. The procedure is described in the following sections.

Creating a D2WComponent

You create a D2WComponent the way you'd create any other component for a regular WebObjects application.

1. In your project, select the WebComponents suitcase.
2. Choose File ▸ New in Project.

   The New File panel opens. Ensure that the Web Components suitcase is open.
3. Type `StudioSelect` for the name of the component.

   A good naming convention for D2WComponents that provide frozen XML is to use the entity name concatenated with the task name. By doing so, components are grouped in Project Builder by entity name.
4. Click OK.

   The WebObjects Component Wizard window opens.
5. Choose None for "Available Assistance" and Java for the Language, and click Finish.
6. Change the component's superclass to D2WComponent.

Edit the **StudioSelect.java** file. First change the component's superclass from WOComponent to D2WComponent. Since D2WComponent is provided by the directtoweb package, you also need to add an import line:

```
import com.apple.yellow.directtoweb.*;
```

Getting the Default XML

After creating your D2WComponent, you need to replace the contents of its **.html** file with the XML it should return to the controller factory. You don't have to create all the XML by hand. Usually you start with the XML that's generated by default and modify that to suit your needs. To get the default XML, you use the Assistant.

1. Start the server and client sides of your application and open the Assistant.

2. Select the XML tab.

   In the form of specifications, the Assistant lists all the requests it can send to the rule system and displays the XML for the selected specification.

   Recall that a specification is a dictionary that specifies all the input to the rule system for it to satisfy a request. A specification includes the request key (under the key "question") and other state information such as task and entity.

   When you select a specification in the XML pane, the Assistant makes the corresponding request to the rule system and then displays the resulting XML.

3. Select the specification for the select Studio dialog.

   The Assistant orders the specifications by entity name, question, and task; so you can easily locate the select Studio dialog specification by scanning the list for the Studio specifications. The specification for the select Studio dialog looks like this:

   ```
   entity = Studio, question = modalDialog, task = select
   ```

   When you select the specification, the Assistant updates the text area to contain the corresponding XML.

4. Save the XML to a file.

Click the "Save To XML" button on the XML pane. A "Save XML" window opens. You only need to copy the XML from this file, so you can save it to any directory with any file name. After copying the XML, you can delete the file.

5. Copy the frozen XML and paste it into your D2WComponent's **.html** file.

Modifying the XML

Once you have the default XML as a starting point, you modify it in two ways:

- First delete the parts of the XML that you don't need to freeze.

  The default XML describes the controller hierarchy for an entire window. You can freeze the XML for the entire window, but you shouldn't if you don't need to. If you let Direct to Java Client generate as much of the window as possible, you can still use the Assistant to configure those parts, which is an advantage.

- Edit the parts of the XML to describe the user interface you want.

Suppose the default XML for the select Studio dialog is the following:

```
<MODALDIALOGCONTROLLER reuseMode="ReuseIfInvisible"
disposeIfDeactivated="false" typeName="question = modalDialog,
task = select, entity = Studio">
    <ACTIONBUTTONSCONTROLLER widgetPosition="Top">
        <QUERYCONTROLLER entity="Studio" minimumWidth="256"
        alignsComponents="true">
            <TEXTFIELDCONTROLLER valueKey="name"
            isQueryWidget="true"/>
            <LISTCONTROLLER minimumWidth="256" editability="Never"
            alignsComponents="true" entity="Studio">
                <TABLECONTROLLER>
                    <TABLECOLUMNCONTROLLER editability="Never"
                    valueKey="name"/>
                    <TABLECOLUMNCONTROLLER valueKey="budget"
                    formatPattern="#,##0.00;-#,##0.00"
                    editability="Never"
                    formatClass=
                    "com.apple.client.foundation.NSNumberFormatter"/>
                </TABLECONTROLLER>
            </LISTCONTROLLER>
        </QUERYCONTROLLER>
    </ACTIONBUTTONSCONTROLLER>
</MODALDIALOGCONTROLLER>
```

To modify the XML, perform the following:

1. Delete the XML for the modal dialog and for the button row.

   To put a box around the **name** query fields, you don't need to freeze the XML for the modal dialog controller (the `MODALDIALOGCONTROLLER` tag) or for the row of buttons (the `ACTIONBUTTONSCONTROLLER` tag). So first delete the first two and the last two lines of the XML.

2. Add a box controller around the text field.

   The following lines add a box controller:

   ```
   <BOXCONTROLLER verticallyResizable="false"
   usesTitledBorder="true" color="255,0,0" font="+0,Italic">
   </BOXCONTROLLER>
   ```

   Put the opening BOXCONTROLLER tag above the lines for the text field controller and the closing BOXCONTROLLER tag below it. This inserts an EOBoxController in the controller hierarchy as a subcontroller of the EOQueryController and the supercontroller of the EOTextFieldController.

   The XML attributes of the BOXCONTROLLER tag specify that the box should not be vertically resizable, the box should have a title, and that the title's color is red and its font is Italic. The EOBoxController derives its title from its subcontroller, the text field controller. In this case, the box uses the text field controller's property, **name**. For more information on how the box controller works, see the EOBoxController class specification in the *EOGeneration Framework Reference*.

3. Remove the label from the text field.

   Because you configured the box to have a title, the text field doesn't need a label. To remove the label, change the TEXTFIELDCONTROLLER line to the following:

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

When you're done with the changes, the resulting XML should be:

```
<QUERYCONTROLLER entity="Studio" minimumWidth="256"
alignsComponents="true">
```

```
    <BOXCONTROLLER usesTitledBorder="true"
    verticallyResizable="false" color="255,0,0" font="+0,Italic">
        <TEXTFIELDCONTROLLER isQueryWidget="true" valueKey="name"
        usesLabelComponent="false"/>
    </BOXCONTROLLER>
    <LISTCONTROLLER minimumWidth="256" editability="Never"
    alignsComponents="true" entity="Studio">
        <TABLECONTROLLER>
           <TABLECOLUMNCONTROLLER editability="Never"
           valueKey="name"/>
               <TABLECOLUMNCONTROLLER valueKey="budget"
               formatPattern="#,##0.00;-#,##0.00"
               editability="Never"
               formatClass=
               "com.apple.client.foundation.NSNumberFormatter"/>
        </TABLECONTROLLER>
    </LISTCONTROLLER>
</QUERYCONTROLLER>
```

Writing a Custom Rule to Use Your Component

Once you've got a component that provides your customized XML, you need a custom rule that tells the rule system to use your component. The rule looks like this:

- Condition (Lhs): ((task = 'select') and (entity.name = 'Studio'))
- Key (Rhs key): controller
- Value (Rhs value): StudioSelect (the name of the D2WComponent subclass that provides the custom XML)
- Priority: 50

This rule is fired when the controller factory tries to generate the EOQueryController for the select Studio dialog. By default, the rule system would use numerous D2WComponents to generate the XML for the query controller. But with this custom rule, the rule system uses your custom StudioSelect D2WComponent to provide the frozen XML.

Freezing Nib Files

Freezing nib files provides even more control over an application's user interface than freezing XML does. Using this technique, you create your own nib files in Interface Builder and instruct Direct to Java Client to use your nibs to create the user interface instead of dynamically generating it.

To create the nib, use Project Builder's "New in Project" command to add a new nib file to the ClientSideJava subproject. In the WebObjects Java Client Interface Wizard, choose EOF Application Skeleton. In your new nib, set the file's owner class to the name of the controller class that loads the nib. Also, be sure that the nib file's editing context and display group are connected.

EOEntityController introduces the ability to create a user interface from a nib file, so you can use frozen nib files only with EOEntityController subclasses (typically EOFormController, EOListController, EOQueryController, and sometimes custom subclasses). To tell an entity controller to use a nib file instead of dynamic generation, you use the entity controller's XML attribute **archiveName**.

When you use a frozen nib file, the application doesn't need the same controllers as it does when it dynamically generates the user interface. Since the nib file describes all of the widgets in the user interface, the application doesn't need any controllers whose sole purpose is to create and configure widgets. For example, EOTextFieldControllers aren't necessary because they do nothing more than create and configure a text field widget. Since the nib file takes over this responsibility, the controller hierarchy corresponding to a frozen nib shouldn't have any EOTextFieldControllers. It is, however, possible to use subcontrollers to generate portions of a user interface. Their components are added to the component of the controller in the usual way.

Remember that freezing interface files has many disadvantages: maintenance is more difficult, platform specific layout is harder to achieve, model changes aren't automatically incorporated, and so on. Therefore, carefully weigh the cost of freezing nib files against the benefits before using this approach.

Adding Custom Actions

Direct to Java Client provides an easy way to add new actions to an application's main menu. As an example of how to do this, this section describes adding an "About WebObjects" window that is activated by a command in the Tools menu.

Adding an Additional Action

Direct to Java Client uses EOAction objects to describe the actions associated with menu commands and buttons.

The default Direct to Java Client rules provide hooks for adding to an application's actions—the commands available in the application's main menu. Recall that when an application starts up, it makes a rule system request with the request key **availableActions**. By default, the system populates an application's menu with commands to Quit, open the Query Window, and so on.

To add custom commands to the menu, you create a rule with the key **additionalActions**. The value should be a D2WComponent that generates XML to describe an EOAction. When the **availableActions** rule is fired, any rule with the request key **additionalActions** is also fired, and the resulting EOAction is added to the set of default actions.

To add the "About WebObjects" command to the Tools menu, do the following:

1. Write a custom rule that specifies the following:

- Condition (Lhs): empty (the rule should always fire)
- Request key (Rhs key): **additionalActions**
- Value (Rhs value): AboutWebObjectsActions
- Priority: 50

The value, AboutWebObjectsActions is a D2WComponent that generates the XML for the custom action.

2. Write the AboutWebObjectsActions D2WComponent

AboutWebObjectsActions.html

```
<WEBOBJECT name=aboutWebObjectsAction/>
```

AboutWebObjectsActions.wod

```
aboutWebObjectsAction: EOSwitchComponent {
    componentName = "EOHelpWindowAction";
    d2wContext = localContext;
    task = "aboutWebObjects";
    multipleWindowsAvailable = "false";
}
```

Creating a Corresponding Window

The **AboutWebObjectsActions.wod** file defines a new task, aboutWebObjects. When the switch component for the **aboutWebObjectsAction** makes its rule system request, it adds an entry to the rule system's D2WContext specifying the

task as aboutWebObjects. In other words, when a user clicks the “About WebObjects” command, the rule system knows that the current task is aboutWebObjects. So, to open the corresponding window, you simply write a custom rule whose condition specifies the task as aboutWebObjects:

1. Write a custom rule that specifies the following:

- Condition (Lhs): task = ‘aboutWebObjects’
- Request key (Rhs key): window
- Value (Rhs value): AboutWebObjects
- Priority: 50

   The value, AboutWebObjects, is a D2WComponent that generates the XML for the custom window.

2. Write the AboutWebObjects D2WComponent

   AboutWebObjects.html

```
<DIALOGCONTROLLER disposeIfDeactivated="true"
label="About WebObjects" reuseMode="AlwaysReuse"
typeName="aboutWebObjects">
   <STATICICONCONTROLLER canResizeHorizontally="false"
   canResizeVertically="false" iconName="PoweredByWebObjects"/>
</DIALOGCONTROLLER>
```

   For this simple example, the corresponding **AboutWebObjectsActions.wod** is empty.

(Optional) Specifying Additional Available Specifications

Recall that when an application starts up, it makes a rule system request with the request key **availableSpecifications**. The result is a list of all the rule system requests the client can make. The requests are expressed as EOSpecification objects, which identify a request key and state information. For more information on specifications, see the section “Server Side XML Generation” (page 47).

The default Direct to Java Client rules provide hooks for adding to an application’s available specifications. The process is similar to that for adding to an application’s available actions. You create a rule with the key **additionalAvailableSpecifications**. The value should be a D2WComponent that generates XML to describe an EOSpecification. When the

availableSpecifications rule is fired, any rule with the request key **additionalAvailableSpecifications** is also fired, and the resulting EOSpecification is added to the set of default specifications.

1. Write a custom rule that specifies the following:

- Condition (Lhs): empty (the rule should always fire)
- Request key (Rhs key): additionalAvailableSpecifications
- Value (Rhs value): AboutWebObjectsSpecifications
- Priority: 50

The value, AboutWebObjectsSpecifications is a D2WComponent that generates the XML for the custom action.

2. Write the AboutWebObjectsSpecifications D2WComponent

AboutWebObjectsSpecifications.html

```
<WEBOBJECT name=aboutWebObjectsSpecification/>
```

AboutWebObjectsSpecifications.wod

```
aboutWebObjectsSpecification: EOSwitchComponent {
    componentName = "EOSpecification";
    question = "window";
    task = "aboutWebObjects";
}
```

This task is optional, but it usually improves performance. It delays the rule system request for the XML description of the window until the window is really needed (the "About WebObjects" command has been clicked). If you don't specify an additional available specification, the client application makes a rule-system request when the command for the action is created. It requests the XML simply to verify that the window is actually present. However, since all windows described in the available specifications are assumed to be accessible, the rule system request is delayed until the window is opened by the end user.

XML Tags and Attributes for EOActions

Most of the action classes used by Direct to Java Client are private. However, you can still use them to add custom actions. The following table describes the

types of actions provided by Direct to Java Client and the corresponding XML tags.

Action	XML Tag	Description
Insert	INSERTACTION	Opens a window for inserting an object of a particular entity. The corresponding command is added to the Document menu.
Open	OPENACTION	Opens a window for an object of a particular entity. The corresponding command is added to the Document menu.
Query	QUERYACTION	Opens a query window for searching for objects of a particular entity. The corresponding command is added to the Document menu.
Help Window	HELPWINDOWACTION	Activates a window for a particular task. The corresponding command is added to the Help menu.
Tool Window	TOOLWINDOWACTION	Activates a window for a particular task. The corresponding command is added to the Tools menu.
Window	WINDOWACTION	Activates a window for a particular task.
Application	APPLICATIONACTION	An action that is sent to the application object.
Controller Hierarchy	CONTROLLERHIERARCHYACTION	An action that is dispatched to controller hierarchy.

Implementing Custom Controller Classes

You can implement your own subclasses of EOController to use on the client side. To make your class known to Direct to Java Client, use the XML attribute **className**. You can specify your class with custom rules or by changing frozen XML. For example, the following frozen XML substitutes the CustomTextFieldController class for EOTextFieldController:

```
<TEXTFIELDCONTROLLER valueKey="name"
className="customapplication.client.CustomTextFieldController"/>
```

The EOXMLUnarchiver maps the XML tag TEXTFIELDCONTROLLER to the EOTExtFieldController by default. Specifying your custom text field controller in the **className** attribute tells the EOXMLUnarchiver to use your text field instead.