
EOModeler User Guide

(Legacy)



2006-05-23



Apple Inc.
© 2002, 2006 Apple Computer, Inc.
All rights reserved.

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

The Apple logo is a trademark of Apple Inc.

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

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

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

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

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

Enterprise Objects is a trademark of Apple Inc.

Java is a registered trademark of Oracle and/or its affiliates.

Times is a registered trademark of Heidelberger Druckmaschinen AG, available from Linotype Library GmbH.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION,

EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

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

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

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

Contents

Introduction Introduction to EOModeler User Guide 11

About This Book 11
Organization of This Document 12
See Also 12

Chapter 1 Data Modeling and EOModeler 13

Why Model Your Data? 13
When to Model Data 13
EOModeler Features 14
Entity-Relationship Modeling Fundamentals 15
 Entities and Attributes 15
 Naming Conventions 15
 Data Types 16
 Relationships 16
 ER Modeling in Enterprise Objects 16
Creating a Model From an Existing Data Source 17
 Selecting an Adaptor 17
 Choosing What to Include 20
 Choosing the Tables to Include 22
 Specifying Primary Keys 22
 Specifying Referential Integrity Rules 22
 Choosing Stored Procedures 23
 Save the Model 23
What a New Model Includes 24
Checking for Consistency 25

Chapter 2 Using EOModeler 27

Editing Views 27
The Tree View 28
Table Mode 29
Diagram View 31
Browser Mode 32

Chapter 3 Working With Attributes 35

Attribute Characteristics 35
More About Attribute Characteristics 37
 Allows Null 38
 Class Property 38

- Client-Side Class Property 38
- Definition (Derived Attributes) 38
- Locking 39
- Primary Key 40
- Read Format and Write Format 40
- Value Type 41
- Prototype Attributes 43
 - Creating Prototype Attributes 43
 - Assigning a Prototype to an Attribute 43
- Flattened Attributes 44
 - When Should You Flatten Attributes? 44
 - Flattening an Attribute 45

Chapter 4 Working With Relationships 47

- About Relationships 47
 - Directionality 47
 - Cardinality 48
 - Relationship Keys 48
 - Reflexive Relationships 49
 - Owns Destination and Propagate Primary Key 50
- Creating Relationships 50
 - Forming Relationships in the Diagram View 50
 - Forming Relationships in the Inspector 51
 - Forming Relationships Across Models and Data Sources 52
- Tips for Specifying Relationships 53
- Adding Referential Integrity Rules 54
 - Optionality 54
 - Delete Rule 54
- Flattened Relationships 55
 - When Should You Flatten Relationships? 55
 - Flattening a Relationship 55
- Modeling Many-to-Many Relationships 57

Chapter 5 Working With Entities 59

- Entity Characteristics 59
- Advanced Entity Inspector 60
- Shared Objects Inspector 62
- Stored Procedure Inspector 63

Chapter 6 Modeling Inheritance 67

- Deciding to Use Inheritance 67
- Vertical Mapping 69
 - Implementing Vertical Mapping in a Model 70

- Advantages of Vertical Mapping 74
- Disadvantages of Vertical Mapping 74
- Horizontal Mapping 74
 - Implementing Horizontal Mapping in a Model 75
 - Advantages of Horizontal Mapping 76
 - Disadvantages of Horizontal Mapping 76
- Single-Table Mapping 76
 - Implementing Single-Table Mapping in a Model 77
 - Implementing a Restricting Qualifier 78
 - Advantages of Single-Table Mapping 79
 - Disadvantages of Single-Table Mapping 79

Chapter 7 Working With Fetch Specifications 81

- Creating a Fetch Specification 81
 - Building a Qualifier 82
 - Creating Compound Qualifiers 83
 - Using Qualifier Variables 84
- Assigning a Sort Ordering 85
- Prefetching 86
- Configuring Raw Row Fetching 86
- Other Fetch Specification Options 87
- Using Named Fetch Specifications 88

Document Revision History 91

Glossary 93

Figures, Tables, and Listings

Chapter 1 **Data Modeling and EOModeler 13**

Figure 1-1	Choose an adaptor 18
Figure 1-2	JDBC Connection window with OpenBase connection information 19
Figure 1-3	JDBC Connection window with Oracle connection information 19
Figure 1-4	JNDI Connection window 20
Figure 1-5	Choose what to include 20
Figure 1-6	Select tables to include in model 22
Figure 1-7	Specify referential integrity rules for a relationship 23
Figure 1-8	Consistency-checking options 25
Table 1-1	Object-relational mapping 17
Table 1-2	Table name to entity name mapping 24
Table 1-3	Column name to attribute name mapping 24

Chapter 2 **Using EOModeler 27**

Figure 2-1	An attribute's inspector panes 28
Figure 2-2	Tree view with an expanded entity 29
Figure 2-3	Table mode with the entire model selected 30
Figure 2-4	A model's components in table mode 30
Figure 2-5	An entity's attributes and relationships 31
Figure 2-6	Diagram view 32
Figure 2-7	Browser mode 33

Chapter 3 **Working With Attributes 35**

Figure 3-1	An entity's attributes 35
Figure 3-2	Derived attribute syntax 39
Figure 3-3	A prototype entity 43
Figure 3-4	An attribute using a prototype 44
Figure 3-5	An attribute using part of a prototype 44
Figure 3-6	Selecting the relationship in which the attribute to flatten exists 45
Figure 3-7	An attribute flattened 46
Figure 3-8	A flattened attribute in the Attribute Inspector 46
Table 3-1	Attribute characteristic definitions 36

Chapter 4 **Working With Relationships 47**

Figure 4-1	Foreign key for PersonPhoto in Person table 48
Figure 4-2	PersonPhoto primary key in PersonPhoto table 48
Figure 4-3	Reflexive relationship table 49

Figure 4-4	Control-drag from source key to destination key to form a relationship	50
Figure 4-5	Control-dragging also creates an inverse relationship	51
Figure 4-6	Using the Relationship Inspector to build a relationship	52
Figure 4-7	Multiple models to choose from	53
Figure 4-8	Select the relationship that contains the relationship to flatten	56
Figure 4-9	Select the relationship to flatten	56
Figure 4-10	A flattened relationship displayed in browser mode	56
Figure 4-11	Two entities before joining in a many-to-many relationship	57
Figure 4-12	Two entities after being joined in a many-to-many relationship	58

Chapter 5 Working With Entities 59

Figure 5-1	The Administrator entity selected in the tree view	59
Figure 5-2	The Advanced Entity Inspector	61
Figure 5-3	Shared Objects Inspector configured to perform an unqualified fetch	62
Figure 5-4	Shared Objects Inspector configured to shared objects from a qualified fetch	63
Figure 5-5	Stored Procedure Inspector	64
Figure 5-6	Map stored procedure in model to procedure in data source	65
Figure 5-7	Stored procedure arguments	65
Table 5-1	Entity characteristics	60

Chapter 6 Modeling Inheritance 67

Figure 6-1	A simple object hierarchy	68
Figure 6-2	Vertical mapping	70
Figure 6-3	Inherited attributes appear in italics	71
Figure 6-4	Mark parent entities as abstract if they won't ever be instantiated	72
Figure 6-5	To-one relationships to parent entity shown in inspector	73
Figure 6-6	Flattened attributes in table view	73
Figure 6-7	Vertical inheritance hierarchy in diagram view	74
Figure 6-8	Horizontal inheritance mapping	75
Figure 6-9	Single-table inheritance mapping	77
Figure 6-10	Assign a restricting qualifier	78
Listing 6-1	Set type in <code>awakeFromInsertion</code>	79

Chapter 7 Working With Fetch Specifications 81

Figure 7-1	A sample fetch specification	82
Figure 7-2	Static qualifier	83
Figure 7-3	A compound qualifier	84
Figure 7-4	Fetch specification bindings in WebObjects Builder	85
Figure 7-5	Sort ordering	85
Figure 7-6	Configure prefetching	86
Table 7-1	Bindings dictionary	88
Listing 7-1	Get a fetch specification programmatically	88

Listing 7-2 Bind qualifier variables 88

Introduction to EOModeler User Guide

Important: The information in this document is obsolete and should not be used for new development. Links to downloads and other resources may no longer be valid.

Note: This document was previously titled *WebObjects EOModeler User Guide*.

The Enterprise Object technology brings the benefits of object-oriented programming to database application development. You can use Enterprise Objects to build feature-rich database applications that encapsulate your business logic yet are independent of any particular data source.

One of the most significant problems developers face when using object-oriented programming languages with relational databases is the difficulty of matching relational database tables with the flexibility afforded by objects.

The Enterprise Object technology solves this problem by providing tools for defining an object model and mapping it to a data model. This allows you to create objects that encapsulate both the data and the methods for operating on that data, while taking advantage of the data-access services provided by Enterprise Objects.

This book teaches you how to use EOModeler to build the data models you need to use Enterprise Objects. With EOModeler, you can build data models based on existing data sources or you can build data models from scratch, which you then use to create data structures (tables, columns, joins) in a data source.

About This Book

This book is an in-depth guide on how to use EOModeler. It does not provide an introduction to the Enterprise Object technology or to WebObjects. Instead, it is meant as supplement to the other introductory WebObjects documentation and also as a general reference guide to EOModeler.

Some features of EOModeler, such as schema synchronization, SQL generation, and Java class file generation are not discussed in this book. These features are best understood in the context of a specific application and so are discussed in the Inside WebObjects series books *Web Applications* and *Java Client Desktop Applications*.

This book assumes some familiarity with relational databases and with Enterprise Objects. If you're new to WebObjects and Enterprise Objects, it is recommended that you start with one of the tutorial-based books, either *Web Applications* or *Java Client Desktop Applications* depending on the kind of application development you are doing. These books provide conceptual introductions to Enterprise Objects and provide contexts in which to best learn about the technology.

Then, after you've read through the introductory material, you'll probably have questions about advanced data modeling techniques or how to use the advanced features of EOModeler. At that point, you are ready for the information in this book.

Organization of This Document

The book includes these chapters:

- [“Data Modeling and EOModeler”](#) (page 13) provides an introduction to data modeling concepts and introduces the data modeling tool provided by WebObjects, called EOModeler.
- [“Using EOModeler”](#) (page 27) introduces the major user interface elements of EOModeler and teaches you how to use the application.
- [“Working With Attributes”](#) (page 35) teaches you how to work with entity attributes in EOModeler. It describes how to configure attribute characteristics, how to use prototype attributes, and how to flatten attributes.
- [“Working With Relationships”](#) (page 47) provides an introduction to relationships between entities. It teaches you how to add and configure relationships in EOModeler, how to specify referential integrity rules in relationships, how to flatten relationships, and how to configure many-to-many relationships.
- [“Working With Entities”](#) (page 59) teaches you how to work with entities in EOModeler. It describes how to configure entity characteristics in EOModeler, how to configure entities for use in a shared editing context, and how to work with stored procedures in EOModeler.
- [“Modeling Inheritance”](#) (page 67) introduces entity inheritance in Enterprise Objects. It describes the three types of entity inheritance and how to model each type using EOModeler.
- [“Working With Fetch Specifications”](#) (page 81) describes how to configure fetch specifications in EOModeler and how to use these fetch specifications programmatically.

See Also

You can find further documentation for WebObjects and Enterprise Objects in three places:

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

Data Modeling and EOModeler

This chapter introduces the concept of **data modeling**. It also introduces the WebObjects data modeling tool, called EOModeler. It then leads you through the process of building a simple model. It is divided into the following sections:

- [“Why Model Your Data?”](#) (page 13) addresses that question.
- [“When to Model Data”](#) (page 13) suggests where in the development cycle data modeling should occur.
- [“EOModeler Features”](#) (page 14) discusses the tool WebObjects provides to model data.
- [“Entity-Relationship Modeling Fundamentals”](#) (page 15) introduces some of the main concepts of this data modeling paradigm.
- [“Creating a Model From an Existing Data Source”](#) (page 17) leads you through the process of creating a simple model.
- [“What a New Model Includes”](#) (page 24) describes what you get from a new model.
- [“Checking for Consistency”](#) (page 25) discusses EOModeler’s consistency-checking feature.

Why Model Your Data?

While it may seem obvious, data modeling is perhaps the most important phase of WebObjects application development. Data models form the foundation of your business logic and business logic forms the core of your application. Good business logic is essential to building effective applications, so it follows that good data models are essential to the success of the applications you build. Most importantly, data modeling plays a crucial role in object-relational mapping, the process in which database records are transposed into Java objects.

Using data models to develop data-driven applications provides you a unique advantage: applications are isolated from the idiosyncrasies of the data sources they access. This separation of an application’s business logic from database logic allows you to change the database an application accesses without needing to change the application.

Also, WebObjects provides a rapid development environment that enables the creation of full-featured Web and desktop applications based on data models. So if you build well-designed data models, WebObjects gives you usable applications for free without any code.

When to Model Data

Whenever possible, data modeling should occur in the earliest phases of application design and development. In data-driven applications, data modeling almost always influences an application’s goals. That is, you cannot define an application’s goals outside of the context of the application’s data models.

Data models in Enterprise Objects reflect the earliest decisions about what data users see and how they are allowed to interact with data. This is because an important part of data modeling is deciding which entities and attributes are visible to clients and editable by clients. So by including data modeling as an early part of the design process, you can simplify future implementation details.

This doesn't imply, however, that you need to lock down a model before you begin other phases of application development. Although it's easier to start with a model that is fairly complete, you can revise models after you've started other phases of application development.

EOModeler Features

WebObjects provides you with a great tool for object-relational mapping called EOModeler. It allows you to

- build data models either from scratch or by analyzing preexisting data sources using reverse engineering
- add and customize entities (tables) and attributes (columns)
- form relationships between entities
- form relationships across multiple models
- generate SQL from a model to create or update a data-source schema based on the model
- generate Java classes from a model in which you can add custom business logic
- use stored procedures within data models
- graphically build fetch specifications for retrieving data
- flatten attributes and relationships
- define derived attributes
- build database queries in raw SQL
- synchronize changes made in the data source schema or in the data model using schema synchronization

How you use EOModeler greatly depends on the control you have over an application's data sources. For instance, if you are not allowed to define new database tables or edit database columns, you have less control over modeling the data; instead your responsibility is to define parameters for object-relational mapping.

You'll create better models if you're empowered to change the database schema, but you can still create effective models if you can't.

The most important thing EOModeler does is to provide a foundation with which to define the mapping between data source schemas and Java objects. EOModeler produces many things, the most important being EOModel files that provide the concrete mapping definition. The industry-standard term for the mapping between data-source schemas and objects is **Entity-Relationship modeling**.

Entity-Relationship Modeling Fundamentals

A data source stores data in the structures it defines: A relational database uses tables, an object-oriented database uses objects, a file system uses files, and so on. Entity-Relationship modeling provides terminology and methodology to describe how a data source's data structures can be mapped to Java objects or other multidimensional data structures. Enterprise Objects uses a modified version of Entity-Relationship modeling.

When working with relational-database systems, you can use EOModeler to define the mapping between relational-database data structures and Java objects. The product of EOModeler, a model file, describes the database's data structures in terms that the Enterprise Object technology can understand and work with.

In an Entity-Relationship model, distinguishable things are known as **entities**, each entity is defined by its component **attributes**, and the affiliations, or **relationships** between entities, are identified (together, attributes and relationships are known as **properties**). From these three simple modeling objects—entities, attributes, and relationships—arbitrarily complex systems can be modeled.

Entities and Attributes

Entities and attributes represent structures that contain data. In a relational database data model, entities represent tables and an entity's attributes represent columns.

Each row in the table can be thought of as an instance of an entity. So, a customer record is called an instance of the Customer entity. Each instance of an entity typically maps to one object, but more complex mappings are possible.

Contained within an entity is a list of attributes of the thing that's being modeled. The Customer entity could contain attributes such as a customer's first name, last name, phone number, and so on.

In traditional Entity-Relationship mapping, each entity represents all or part of one database table. However, Enterprise Objects allows you to go beyond this by adding attributes to an entity that actually reflect data in other related tables (this type of attribute is known as a **flattened attribute**). So, in Enterprise Objects, an entity can map to one or more database tables.

Entities can also have **derived attributes**, which do not directly correspond to any of the columns in the database table to which the entity maps. These attributes are usually computed from one or more attributes in the entity. For example, a derived attribute could represent the total amount of money spent by a customer by adding the amounts of all a customer's purchases together.

Naming Conventions

Entities and attributes in data models derive their names based on the database elements they represent. A CUSTOMER table in a database, for example, would likely map to an entity named "Customer," though this is not strictly required. In your application, you refer to the CUSTOMER table not by its database name but by using its entity name, Customer.

This allows you to build data models that are largely independent of any particular data source. Different data sources enforce different naming conventions for their data structures, so providing a mapping between those names allows you to build reusable data models that can be used with different data sources.

End users don't care if the Customer entity maps to a table named CUSTOMER or CUST or _CUSTOMER, so the mapping also helps make the earliest decisions about how data structures appear in the end-user application.

Data Types

Every database column is assigned a data type such as `int`, `varchar`, `float`, or `blob`. These data types map to primitive Java types such as `int` and `float` but also to Java objects such as `java.lang.String` and `java.lang.Object`. The data model defines the data-type mapping between columns and attributes just as it defines the naming mapping between them.

When data is fetched from a data source into an enterprise object, the value of each attribute is converted from its external data type into a Java-specific data type. Likewise, when the data in an enterprise object is pushed back to a data source, each attribute is converted from its Java data type to the data type specified in the model.

Relationships

In the relational-database paradigm, a relationship expresses the affiliation between two tables in the data source. Relationships allow one table to access related information in another table. In more complex data modeling, relationships can express the affinity between multiple tables.

At the data-source level, the two tables in a relationship are linked together using primary and foreign keys. How the table pair is related (that is, what kind of relationship is expressed) depends on the configuration of primary and foreign keys in each table.

At the Enterprise Objects level, relationships are mapped to a particular kind of object that includes not only information about the tables involved in a particular relationship but also the rules of the relationship between the two tables.

Relationships are a complex concept in the Entity-Relationship model. The chapter [“Working With Relationships”](#) (page 47) provides a more thorough introduction to the concept of relationships and also introduces the additional features you get when using relationships with Enterprise Objects.

ER Modeling in Enterprise Objects

In Enterprise Objects, data models describe the data source-to-enterprise object mapping by using these objects:

- `com.webobjects.eoaccess.EOModel`
- `com.webobjects.eoaccess.EOEntity`
- `com.webobjects.eoaccess.EOAttribute`
- `com.webobjects.eoaccess.EORelationship`

The following table describes the object-relational mapping provided by Enterprise Object models. The column titled “Database/directory element” lists elements as they are named in relational databases and then elements as they are named in LDAP directories, if applicable.

Table 1-1 Object-relational mapping

Database/directory element	Model object	Object mapping
Schema	EOModel	Enterprise object model
Table/object class	EOEntity	Enterprise object class
Row/entry	EOEnterpriseObject	Enterprise object instance
Column/attribute	EOAttribute	Enterprise object instance variable
Referential constraint/foreign key	EORelationship	Reference to another enterprise object

While the modeling classes correspond to elements in a data source, a model represents a level of abstraction above the data source. Consequently, mapping between modeling classes and database components doesn't have to be one-to-one. So, for example, while an EOEntity object described in a model corresponds to a single database table, it can contain references to multiple tables. In that sense, a model is more analogous to a database view.

Similarly, an EOAttribute can either correspond directly to a column in the root entity or it can be derived or flattened. A derived attribute typically has no corresponding database column while a flattened attribute is added to one entity from another entity.

Creating a Model From an Existing Data Source

Although models can be programmatically generated at runtime, you typically create models using EOModeler and then add them to your project as model files or as part of a framework your project includes.

This section describes how to create a new model from an existing data source. It is organized in the following sections:

- [“Selecting an Adaptor”](#) (page 17)
- [“Choosing What to Include”](#) (page 20)
- [“Choosing the Tables to Include”](#) (page 22)

EOModeler is installed in `/Developer/Applications`.

To create a new model, choose **New** from the **Model** menu. EOModeler starts the **New Model Wizard**, which helps you configure the new model.

Selecting an Adaptor

An adaptor is an object that connects your application to a particular data source. Since data sources use different connection protocols (JDBC, JNDI, ERP, and so forth), you need a specialized adaptor for each type of data source you use. Also, since vendors build different features into their data sources, you also need an adaptor plug-in for each data source you use.

In WebObjects 5.2, adaptors are provided for JDBC and JNDI connectivity and adaptor plug-ins are provided for the following data sources:

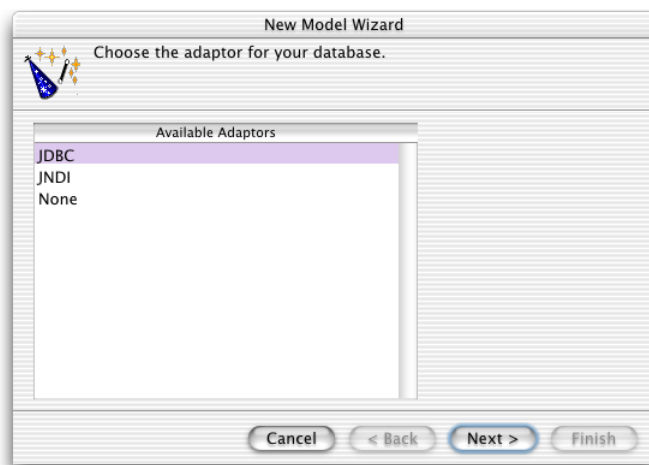
- Oracle (JDBC)
- Microsoft SQL Server 2000 (JDBC)
- MySQL (JDBC)
- OpenBase (JDBC)
- Sybase (JDBC)
- OpenLDAP (JNDI)
- iPlanet (JNDI)

Refer to the document *Post Installation Guide* for exact specifications on which data sources are qualified for WebObjects.

The Enterprise Objects adaptor architecture is modular so you can write adaptor plug-ins for other types of data sources and for other data-source vendors (see Apple Technical Note TN2027). However, this is a very advanced feature so it's best to use a data source that's officially supported by WebObjects, such as those listed above.

WebObjects provides two adaptor types out of the box: JDBC and JNDI. The first pane that appears in the New Model Wizard allows you to choose the adaptor you want to use in the model, as shown in Figure 1-1.

Figure 1-1 Choose an adaptor



After you select an adaptor, EOModeler displays a window prompting you to enter login and other information to connect to the data source. Figure 1-2 shows the JDBC Connection window that appears if you select the JDBC adaptor. Figure 1-4 shows the JNDI Connection window that appears if you select the JNDI adaptor. For this example, use the JDBC adaptor.

Note: Although this example assumes that you choose the JDBC adaptor, there is valuable information throughout regarding the JNDI adaptor.

If necessary, supply the correct login information for the data source. You must also supply the URL of the data source. Depending on the data source and adaptor you want to use, you may also need to supply information in the Driver and Plugin fields.

Note: If you are connecting to a database on your local machine and you are not connected to a network, you may need to use 127.0.0.1 as the address rather than `localhost`.

For this example, use the WORealEstate database that is installed with WebObjects. The example databases uses OpenBase, so the URL you enter in the JDBC Connection window is specific to that particular database. When you create models from other vendor's databases, you may need to use a different URL format. [Figure 1-3](#) (page 19) shows the URL format Oracle uses.

Figure 1-2 JDBC Connection window with OpenBase connection information

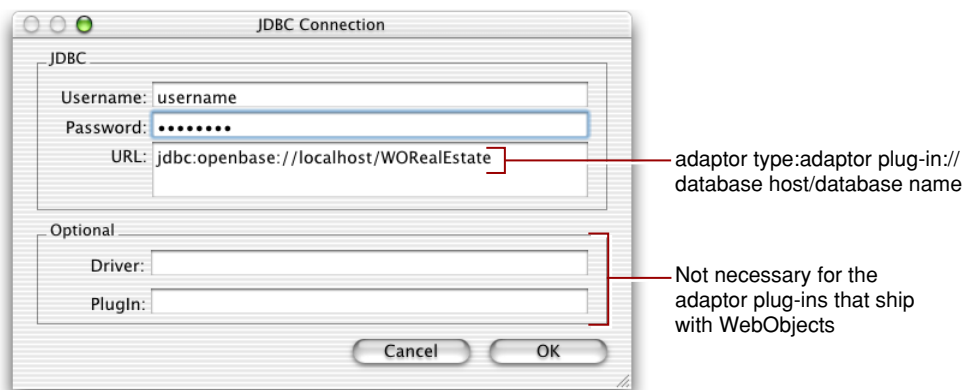


Figure 1-3 JDBC Connection window with Oracle connection information

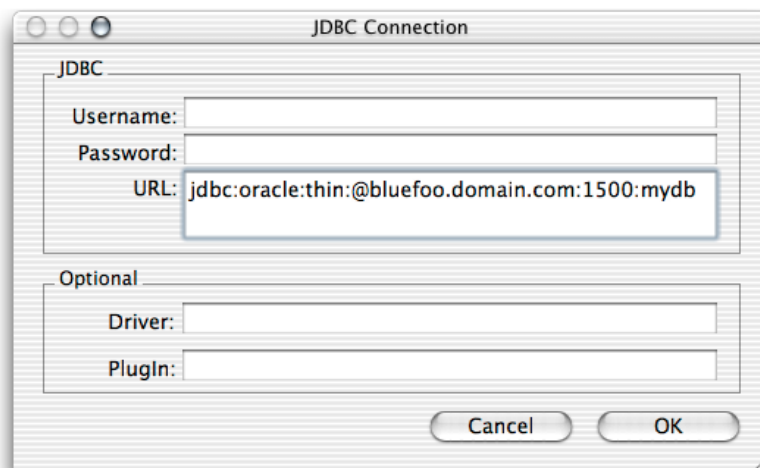
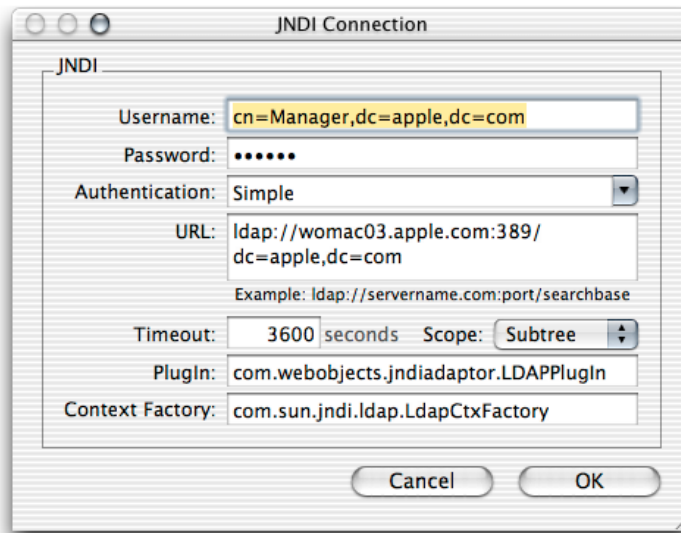
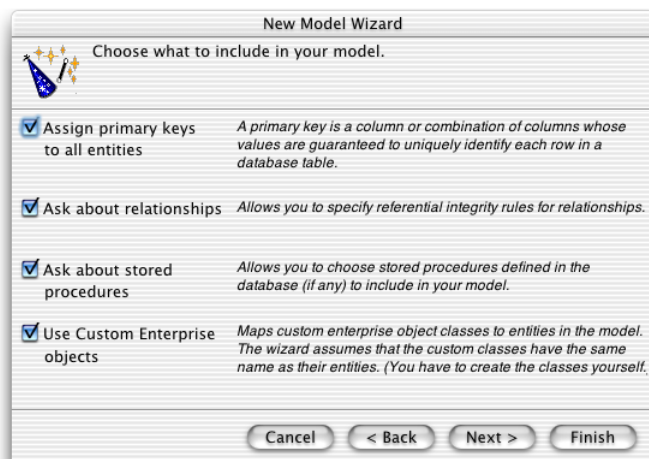


Figure 1-4 JNDI Connection window

Choosing What to Include

After you log in to the data source, the wizard asks you what you want to include in the model, as shown in Figure 1-5. The options available here largely depend on how complete the schema information in the data store is. For example, the wizard includes relationships in your model only if the server's schema information specifies foreign key constraints.

Figure 1-5 Choose what to include

Using the options in this page, you tell EOModeler that you want to supplement the model with additional information. Keep in mind that the wizard does not modify the data source. Also note that you can do everything the wizard does to a model manually after the initial model is created—but using the wizard saves you time.

Note: If you use the JNDI adaptor, always deselect these four options as they don't apply to those types of data sources.

Assign Primary Keys to All Entities

Enterprise Objects uses primary keys to create unique identifiers (`com.webobjects.eocontrol.EOGlobalID` objects) for enterprise objects and also to map enterprise objects to the appropriate database rows. Therefore, every entity in a model must be assigned a primary key.

If you select this option in the New Model Wizard, primary keys are assigned to entities in the model if the wizard finds primary key constraints in the data store's schema information. If the data store's schema does not include primary key information, the wizard first asks you to select the primary key for each entity before it assigns it to an entity in the model.

Ask About Relationships

If a data store's schema includes foreign key constraints, the wizard creates corresponding relationships in the model. However, foreign key definitions in a schema may not provide enough information for the wizard to set all of a relationship's options (such as delete rule and optionality). Selecting this option causes the wizard to prompt you to provide the additional information it needs to configure the relationships, if necessary.

Ask About Stored Procedures

Selecting this option causes the wizard to read stored procedures from the data source's schema information, display them, and allows you to choose which to include in the model. See [“Stored Procedure Inspector”](#) (page 63) for more information.

Use Custom Enterprise Objects

When deciding what class to map a table to, the wizard offers two choices: `EOGenericRecord` or a custom class (a subclass of `EOGenericRecord` or `EOCustomObject`). `EOGenericRecord` is a class that implements the `com.webobjects.eocontrol.EOEnterpriseObject` interface. Its instances store key-value pairs that correspond to an entity's properties and the data associated with each property.

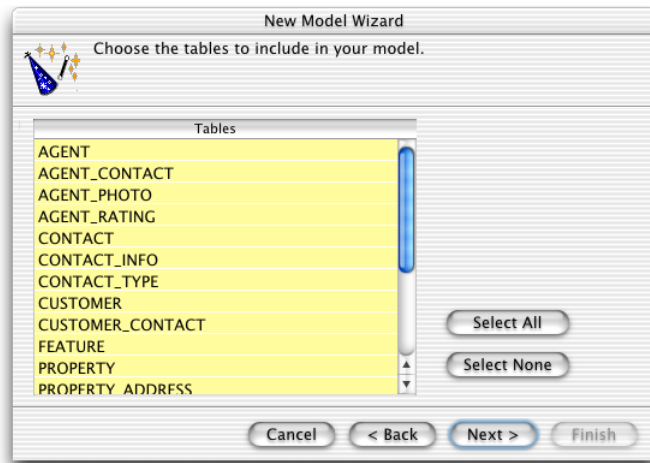
If you do not select this option in the New Model Wizard, the wizard maps all your database tables to `EOGenericRecord`. If you do select this option, the wizard maps all your database tables to custom classes that you write. The wizard assumes that each entity is to be represented by a custom class with the same name. For example, a table named `LISTING` corresponds to an entity named `Listing`, which corresponds to an `EOGenericRecord` or `EOCustomObject` subclass named `Listing.java`.

If you choose to use `EOGenericRecord` rather than custom classes, you do not generate Java files for the entities in your model. This means that you have less opportunity to add business logic to perform validation, insert initial values, or otherwise manipulate enterprise objects programmatically. For some entities, this may be what you want. A common design pattern is to use custom classes only for entities in which you need to implement custom business logic and to use `EOGenericRecord` for other entities.

Choosing the Tables to Include

In this section of the wizard, you are prompted to choose the tables to include in the model. By default all tables are selected, as shown in Figure 1-6, but you can select only certain tables if you want. The wizard creates entities only for the selected tables. Keep in mind that if you want a join to be reflected in the model as a relationship, you must select the source table and the destination table of the join.

Figure 1-6 Select tables to include in model



Specifying Primary Keys

If you are using a data source that stores primary key information in its schema information, the wizard skips this step. It has already read primary key information from the schema and assigned primary keys to entities in the model.

However, if primary key information isn't specified in the data source's schema or if the adaptor can't read it, the wizard asks you to specify a primary key for each entity.

If an entity's primary key is compound; that is, if it's composed of more than one attribute, Command-click to select multiple attributes to use as a compound primary key. You usually use a compound primary key when any single attribute isn't sufficient to identify a row uniquely. That is, you should need to use a compound primary key only if no single attribute can uniquely identify a row.

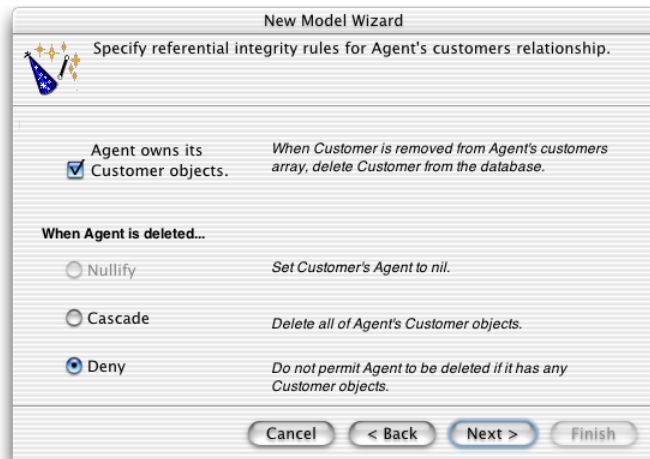
Specifying Referential Integrity Rules

If foreign key constraints aren't specified in the data source's schema information or if the adaptor can't read that information, the wizard doesn't create relationships in the model so it skips this step. If this is the case, you can add relationships later as described in ["Working With Relationships"](#) (page 47).

However, if you're using a data source that stores foreign key constraints in its schema information, the wizard reads them and creates corresponding relationships in the model.

At this point, if you specified that the wizard ask about relationships, it now asks you to provide additional information about the relationships so it can further configure them. Figure 1-7 shows the wizard asking about the `customers` relationship in the Agent entity.

Figure 1-7 Specify referential integrity rules for a relationship



Owns Destination

This option lets you specify whether the relationship's source owns its destination objects. If a source object owns its destination object, for example, as when an Agent object owns its Customer objects, then when something happens so that a destination object (Customer) is removed from the relationship, it is also removed from the data source. Ownership implies that an owned object cannot exist without its owner.

Delete Rule

The options in this section of the window specify what to do when the source object of a relationship is deleted. There are four options, which are described in ["Delete Rule"](#) (page 54).

Choosing Stored Procedures

If you asked the wizard to include stored procedures in your model, in this window it asks you to specify which stored procedures to include. By default, all stored procedures are selected. If you later decide that you want to include a stored procedure in the model that you didn't select here, you can always add it later. See ["Stored Procedure Inspector"](#) (page 63) for more information.

Save the Model

After using the wizard to create the initial model, save the model. If there are any warnings indicating problems with the model, go ahead and continue with the save, then go back and fix the model. You can check for problems at any time by choosing Check Consistency from the File menu.

What a New Model Includes

When you create a new model, the information it includes depends on how complete the underlying data source is and also on the choices you made in the New Model Wizard. EOModeler can read all of the following from a data source and compose a model from it:

- table and column names
- column data types, including the width constraint of string data types
- primary keys
- constraints defined in the database such as “NOT NULL” and “UNIQUE”
- foreign key constraints (which are expressed in models as relationships)
- stored procedures

A model contains not only the information it reads from a data source, but also values it derives from that information, including

- entity and attribute names
- a mapping between the data type of a data column and a corresponding value class (object type), such as `String` (`java.lang.String`), `Number` (`java.lang.Number`), and `Data` (`com.webobjects.foundation.NSData`)

EOModeler derives entity names by taking a data-source table name and making all of it lowercase except for the first letter. It then removes underscore characters and capitalizes the first character following each removed underscore. Table 1-2 gives an example.

Table 1-2 Table name to entity name mapping

Data-source table name	Model entity name
PERSON	Person
PERSON_PHOTO	PersonPhoto
PERSON_REVIEW_NOTES	PersonReviewNotes

Attribute names are based on corresponding database columns. They are derived in the same way as entity names except that EOModeler doesn’t capitalize the first character. EOModeler follows the Java convention for naming methods and instance variables. An example appears in Table 1-3.

Table 1-3 Column name to attribute name mapping

Data-source column name	Model attribute name
NAME	name
LAST_NAME	lastName
FINAL_IMAGE_NAME	finalImageName

Although default entity and attribute names follow a naming convention, you can choose to use arbitrary names. All that really matters is that the external names of the entities match the table names in the schema and that the external names of the attributes match the column names in the tables. However, since business objects are Java objects, your code will be more readable and manageable if you follow the default naming convention for entities and attributes which follows standard Java naming conventions.

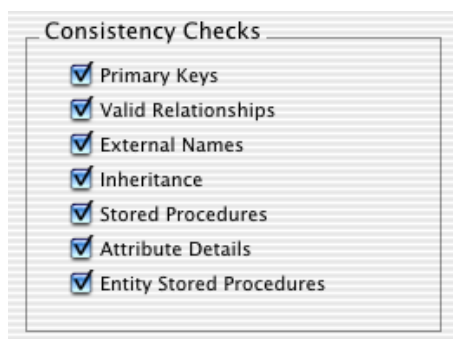
Checking for Consistency

EOModeler provides consistency checking to confirm that your model is valid. For example, a model that has entities without primary keys or relationships without joins is not valid.

You can check your model at any time by choosing Check Consistency from the Model menu. Consistency checking is also invoked automatically whenever you save a model. When a consistency check occurs and inconsistencies are found, the Consistency Check window appears with a list of diagnostic messages.

You can choose what consistency checking looks for in EOModeler's Preferences pane, as shown in Figure 1-8.

Figure 1-8 Consistency-checking options



Using EOModeler

This chapter describes the basic views and major user interface elements of EOModeler. It's organized in the following sections:

- [“Editing Views”](#) (page 27) introduces the various editors in EOModeler.
- [“The Tree View”](#) (page 28) describes how to navigate a model's entities and relationships in the tree view.
- [“Table Mode”](#) (page 29), [“Diagram View”](#) (page 31), and [“Browser Mode”](#) (page 32) discuss the three main editing views.

Editing Views

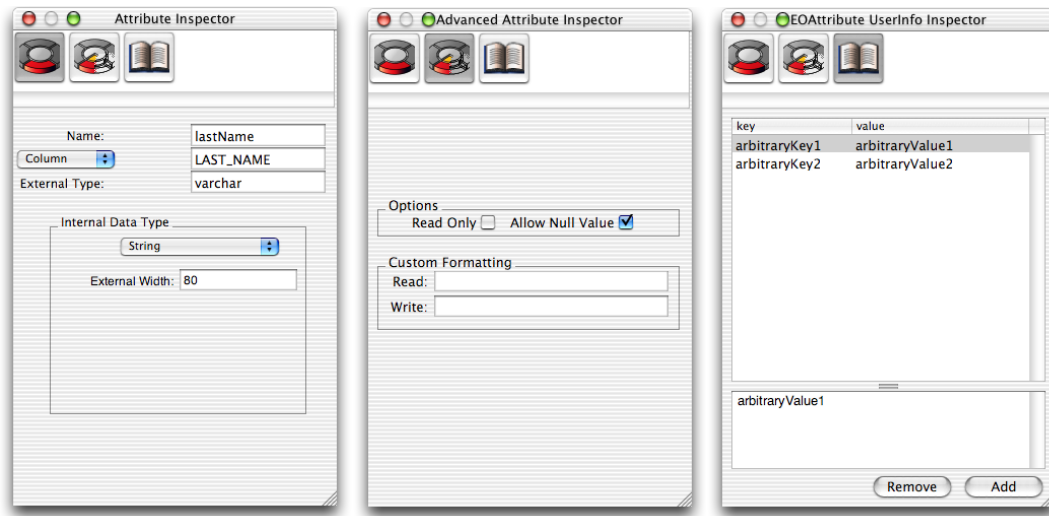
There are a number of different editors in EOModeler. Of these, there are three main views that you spend most time in. Each is appropriate for different uses:

- **Table mode** is appropriate for making basic changes to attributes and attribute characteristics. It is also the view in which you define custom class names for entities.
- **Browser mode** is appropriate for traversing relationship paths and for flattening attributes and relationships.
- **Diagram view** is appropriate for forming relationships between entities and changing certain attribute characteristics. It also provides a graphical view of all a model's elements, including the relationships between entities.

You can switch between display modes with the Change Display View pop-up menu in the toolbar or by choosing Diagram View, Browser Mode, or Table Mode from the Tools menu.

In addition to these views, EOModeler includes a dynamic inspector that changes depending on the element selected. When you select an entity, for example, the inspector becomes the Entity Inspector. Likewise, when you select an attribute, the inspector becomes the Attribute Inspector. Whereas the three main views allow you to edit basic characteristics of a model's elements, an element's inspector provides an interface to configure every characteristic of a particular element.

The inspector for each element itself includes multiple panes. Every selected element has at least three inspector panes: the basic inspector, an Advanced Inspector, and a UserInfo Inspector. Entities have two additional panes: the Stored Procedures Inspector and the Shared Objects Inspector. Figure 2-1 shows the three panes in the inspector when an attribute is selected.

Figure 2-1 An attribute's inspector panes

You use the UserInfo Inspector to add key-value pairs to the UserInfo dictionary. This dictionary provides a mechanism for extending your model. You can use it to define custom behavior for an entity or, more commonly, to add meta-information to a model, an entity, or an attribute. Advanced users sometimes use the UserInfo dictionary when using custom delegates in the access layer.

The Tree View

You navigate a model by selecting entities in the editor's tree view. The root element of the tree view represents the whole model. You can double-click the root element to expand and contract the tree view. When the tree view is expanded, it shows the model's entities. The tree view is always visible in table mode and in diagram view. It is not available in browser mode.

You can also expand and contract a model's entities and stored procedures. Expanding an entity displays the entity's relationships, as shown in Figure 2-2. A relationship in the tree view represents the relationship's destination entity. You can continue to expand the relationship in the tree view to display the destination entity's relationships. Expanding the Stored Procedures folder displays the model's stored procedures.

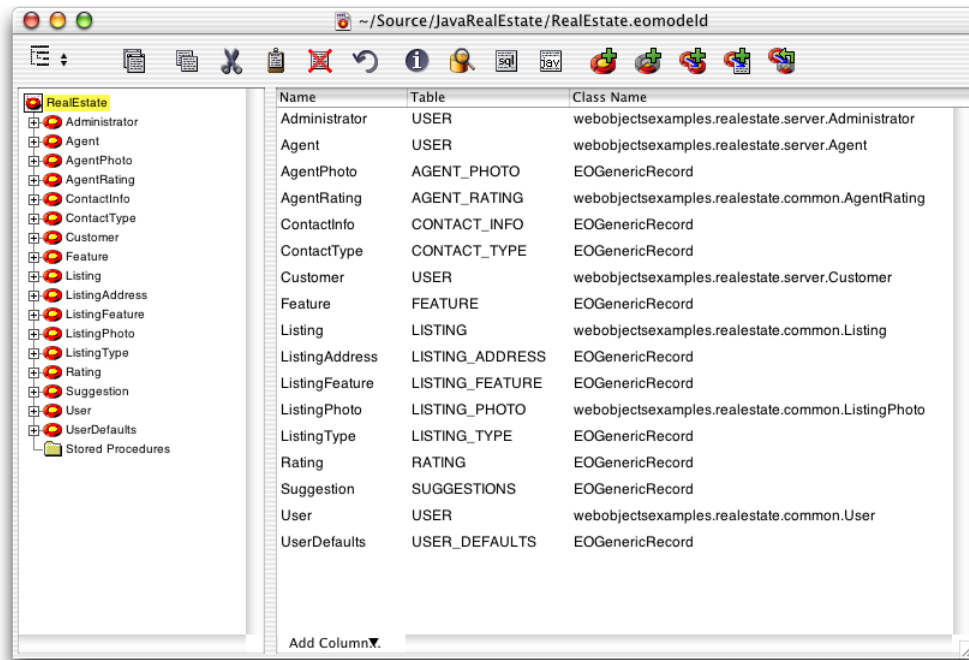
Figure 2-2 Tree view with an expanded entity

The tree view is also useful when performing drag-and-drop operations, such as when dragging an entity or relationship into WebObjects Builder or into Interface Builder. See the tutorials in the books *Inside WebObjects: Web Applications* and *Inside WebObjects: Java Client Desktop Applications* for more information.

Table Mode

The default view mode in EOModeler is table mode. In this mode, EOModeler displays a tree view for viewing a model's entities and relationships within those entities, and a table view whose contents change depending on what's selected in the tree view. You can use table mode to edit many characteristics of entities or of an entity's attributes.

When the root of the tree view is selected, the table view displays the classes with which each entity is associated. When a particular entity is selected in the tree view, the table view changes to display that entity's attributes and relationships.

Figure 2-3 Table mode with the entire model selected

If you switch to another view mode, you can always return to table mode by choosing Table Mode from the Tools menu.

Figure 2-4 shows the table view when the root of the tree view is selected (the root is the name of the model file). The model's entities are displayed, one per row. The columns of the table display information about each entity: the classes with which each entity is associated, whether the entity is read-only, the name of each entity's corresponding database table, and so on.

Figure 2-4 A model's components in table mode

Name	Table	Class Name	Parent
Administrator	USER	webobjectsexamples.realestate.server.Administrator	User
Agent	USER	webobjectsexamples.realestate.server.Agent	User
AgentPhoto	AGENT_PHOTO	EOGenericRecord	
AgentRating	AGENT_RATING	webobjectsexamples.realestate.common.AgentRating	
ContactInfo	CONTACT_INFO	EOGenericRecord	
ContactType	CONTACT_TYPE	EOGenericRecord	
Customer	USER	webobjectsexamples.realestate.server.Customer	User
Feature	FEATURE	EOGenericRecord	
Listing	LISTING	webobjectsexamples.realestate.common.Listing	
ListingAddress	LISTING_ADDRESS	EOGenericRecord	
ListingFeature	LISTING_FEATURE	EOGenericRecord	
ListingPhoto	LISTING_PHOTO	webobjectsexamples.realestate.common.ListingPhoto	
ListingType	LISTING_TYPE	EOGenericRecord	
Rating	RATING	EOGenericRecord	
Suggestion	SUGGESTIONS	EOGenericRecord	
User	USER	webobjectsexamples.realestate.common.User	
UserDefaults	USER_DEFAULTS	EOGenericRecord	

When an entity is selected in the tree view, the table view displays that entity's attributes and relationships, as shown in Figure 2-5. The attributes shown in Figure 2-5 are displayed in italics to indicate that they are inherited from a parent entity. Inheritance is an advanced modeling technique that is discussed in [“Modeling Inheritance”](#) (page 67).

The External Type column represents the data type of the attribute in the data source, such as `varchar`, `long`, and `int`. The Value Class (Java) column represents the object type used when the column is mapped to an enterprise object's attribute. The Value Type column represents how the model's JDBC adaptor plug-in more specifically deals with different object types. See [“Value Type”](#) (page 41) for more details.

Figure 2-5 An entity's attributes and relationships

Agent Attributes									
			0	Name	Column	Value Class (Java)	External Type	Width	Value Type
				<i>firstName</i>	<i>FIRST_NAME</i>	<i>String</i>	<i>varchar</i>		<i>80</i>
				<i>lastName</i>	<i>LAST_NAME</i>	<i>String</i>	<i>varchar</i>		<i>80</i>
				<i>login</i>	<i>LOGIN</i>	<i>String</i>	<i>varchar</i>		<i>20</i>
				<i>password</i>	<i>PASSWORD</i>	<i>String</i>	<i>varchar</i>		<i>20</i>
				<i>userID</i>	<i>USER_ID</i>	<i>Number</i>	<i>long</i>		<i>i</i>
				<i>userType</i>	<i>USER_TYPE</i>	<i>Number</i>	<i>int</i>		<i>i</i>
Add Column▼									

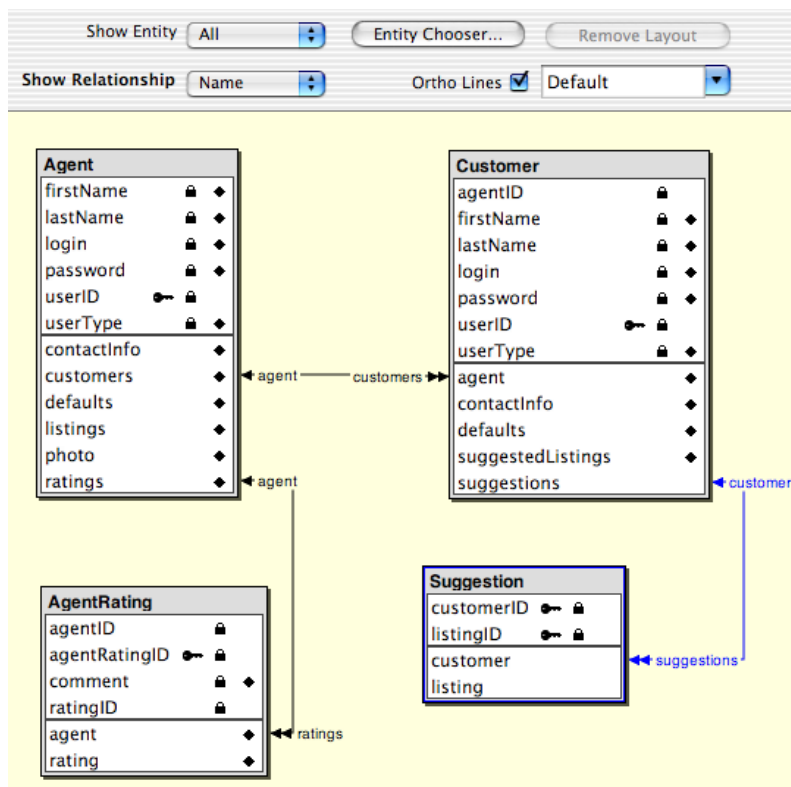
Agent Relationships				
		Name	Destination	Source Att Dest Att
»		<i>contactInfo</i>	<i>ContactInfo</i>	<i>userID</i> <i>userID</i>
»		<i>defaults</i>	<i>UserDefaults</i>	<i>userID</i> <i>userID</i>
»		<i>customers</i>	<i>Customer</i>	<i>userID</i> <i>agentID</i>
»		<i>listings</i>	<i>Listing</i>	<i>userID</i> <i>agentID</i>
>		<i>photo</i>	<i>AgentPhoto</i>	<i>userID</i> <i>agentID</i>
»		<i>ratings</i>	<i>AgentRating</i>	<i>userID</i> <i>agentID</i>

The information you see in the table view when an entity is selected in the tree view depends on the columns visible in the table view. You can add columns by selecting one from the Add Column pop-up menu in the table view. When you add a column to the table, the corresponding menu item is removed from the Add Column pop-up menu. You can remove columns from the table view by selecting a column header and pressing Delete.

Diagram View

In diagram view, you see a visual representation of your model's entities, their attributes, and most important, the relationships between entities, as shown in Figure 2-6. As with table mode, you can use the diagram view to edit components of the model (such as changing attribute names and assigning primary keys to entities) but its editing capabilities are more limited. Diagram view is also useful for printing models.

Figure 2-6 Diagram view



In this view, you can specify which entities are displayed, what kind of attributes are displayed (such as primary keys, class properties, or relationships), and what kind of information about the model's relationships is displayed (such as optionality and propagate primary key) by using the options at the top of the view.

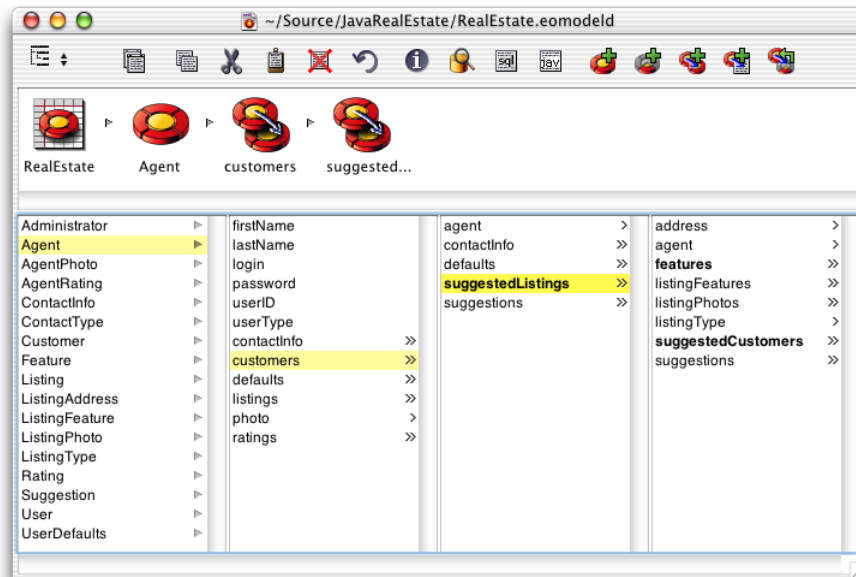
You can also use diagram view to create relationships between entities. By Control-dragging from one relationship key to another, diagram view creates relationships between two entities. Forming relationships this way creates two relationships: one from the source entity to the destination entity and an inverse relationship between the two entities.

Browser Mode

The browser mode is useful for traversing an entity's relationships and also for creating flattened attributes and relationships within an entity. To display the attributes for a particular entity, select the entity in the left-most column while in browser mode. The attributes of that entity then appear in the column to the right of the entity along with the entity's relationships.

Figure 2-7 shows the browser mode traversing the Agent entity, the `customers` relationship in the Agent entity, and the `suggestedListings` flattened relationship in the Customers entity (which is the destination of Agent's `customers` relationship).

Figure 2-7 Browser mode



Working With Attributes

This chapter is divided into the following sections:

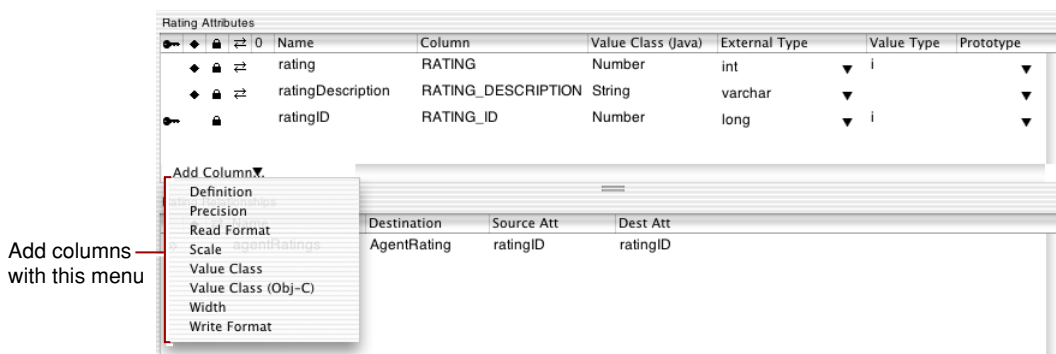
- [“Attribute Characteristics”](#) (page 35) describes all the possible characteristics attributes can have, including value class, value type, and read and write formats. It also describes how to edit each characteristic.
- [“More About Attribute Characteristics”](#) (page 37) provides more detail on certain characteristics like class property, locking, definition (derived attributes), and primary key.
- [“Prototype Attributes”](#) (page 43) discusses what prototype attributes are, how to create them, and how to assign them to attributes.
- [“Flattened Attributes”](#) (page 44) discusses when and how to flatten attributes.

Attribute Characteristics

EOModeler provides three mechanisms for viewing and modifying an entity’s attributes: the table mode of the model editor, the diagram view of the model editor, and the Attribute Inspector. You can use any of the mechanisms to examine the characteristics of your model’s attributes and to make refinements. Each has advantages over the other and is useful in different circumstances.

To display an entity’s attributes in table mode, select an entity in the tree view. Figure 3-1 shows the attributes of an entity called Rating.

Figure 3-1 An entity’s attributes



Each table column corresponds to a single characteristic of the attribute, such as its name, external type, or precision. By default, the columns included in this view represent only a subset of possible characteristics you can set for a given attribute. To add columns for additional characteristics, you use the Add Column pop-up menu in the lower-left corner of the table, as shown in Figure 3-1. You can also resize and rearrange columns in the table.

Table 4-1 describes the characteristics you can set for an attribute. Unless otherwise specified, the instructions are for editing the characteristic in the model editor's table mode. Some of the characteristics are described in more detail in the cross-referenced sections.

Table 3-1 Attribute characteristic definitions

Characteristic	What it is	How you modify it
Allows Null	Indicates whether the attribute can have a <code>null</code> value. See “Allows Null” (page 38).	Click in the column with the checkmark to toggle the option on and off. You can also edit this characteristic in the Attribute Inspector.
Class Property	Indicates that you want to include the attribute in your Enterprise Object classes. See “Class Property” (page 38).	Click in the diamond column to toggle the option on and off. You can also edit this characteristic in diagram view.
Client-Side Class Property	Indicates that you want to include the attribute in your Enterprise Object classes that live on the client side of Java Client applications. See “Client-Side Class Property” (page 38).	Click in the column with the two opposing arrows to toggle the option on and off.
Column	The name of the column in the data source that corresponds to the attribute.	Edit the table cell.
Definition	The SQL definition for a derived attribute. Note that Column and Definition are mutually exclusive; you can't set both. Setting one clears the other. See “Definition (Derived Attributes)” (page 38).	Edit the table cell.
External Type	The data type of the attribute as it's understood by the data source.	Choose a value from the pop-up menu.
Locking	Indicates whether an attribute participates in optimistic locking. See “Locking” (page 39).	Click in the pad-lock column to toggle locking off and on for a particular attribute. You can also edit this characteristic in diagram view.
Name	The name of the attribute as it appears in your application and in your enterprise objects. EOModeler derives a default name based on the corresponding column in the data source which you can edit if necessary.	Edit the table cell. You can also edit this characteristic in diagram view.
Precision	The number of significant digits. The number 12.34 has a precision of four and a scale of 2.	Edit the table cell or use the Attribute Inspector.
Primary Key	Declares whether a property is or is part of the entity's primary key. See “Primary Key” (page 21).	Click in the key column to toggle the primary key off and on. You can also edit this characteristic in diagram view.

Characteristic	What it is	How you modify it
Prototype	A prototype attribute from which this attribute derives its characteristics. See “Prototype Attributes” (page 43).	Choose a value from the pop-up menu. See “Prototype Attributes” (page 43) to learn where these values are defined.
Read Format	The format string that’s used to format the attribute’s value for SQL SELECT statements. In the string, %P is replaced by the attribute’s external name. This string is used whenever the attribute is referenced in a SQL select statement or qualifier. See “Read Format and Write Format” (page 40).	Edit the table cell.
Read Only	Indicates whether the attribute is read only.	Use the Advanced Attribute Inspector.
Scale	The number of digits to the right of the decimal point. Can be negative. Applies only to noninteger, numerical types. The number 12.34 has a scale of 2.	Edit the table cell or use the Attribute Inspector.
Value Class	Not applicable in WebObjects 5.	
Value Class (Java)	The Java type to which the attribute will be coerced in your enterprise objects.	Edit the table cell or use the Attribute Inspector.
Value Class (Obj-C)	Not applicable in WebObjects 5.	
Value Type	The conversion character (such as “i” or “d”) the JDBC adaptor uses to communicate with the data source. See “Value Type” (page 41).	Edit the table cell or use the Attribute Inspector.
Width	The maximum width in bytes or chars of the attribute (applies to string and raw data only).	Edit the table cell or use the Attribute Inspector.
Write Format	The format string that’s used to format the attribute’s value for SQL INSERT or UPDATE expressions. In the string, %V is replaced by the attribute’s external name. For LDAP data sources accessed via the JNDI adaptor, write format specifies the pattern used to generate the relative distinguished name. See “Read Format and Write Format” (page 40).	Edit the table cell.

More About Attribute Characteristics

This section provides information about the more complex characteristics available to attributes.

Allows Null

By default, attributes you add to entities allow `null` values, except in the case of primary keys. This means that Enterprise Objects allows attributes containing no values to be saved to the data source. In some cases, such as when using inheritance, allowing null values may be necessary.

Class Property

When an attribute is marked as a class property, Java classes generated by EOModeler contain accessor methods for that attribute. (However, these Java classes do not contain instance variables for those attributes since the instance data is accessed by the mechanism of key-value coding.) You should mark as class properties only those attributes that are useful in your business logic. This reduces the amount of code to maintain and makes your enterprise object classes more readable.

Primary keys and foreign keys should not be marked as class properties. This is for two reasons: Enterprise objects have no knowledge of the primary and foreign keys of the tables from which they are mapped, and these keys are of no use to your business logic. Also, to ensure that the automatic primary key generation feature of Enterprise Objects is properly invoked, primary keys must not be marked as class properties.

In the process of building enterprise objects, if you find that you need access to primary or foreign keys, there are utility classes and methods that allow you to access these keys even when they are not marked as class properties. See the API reference in the `com.webobjects.eoaccess` package for `EOEntity.primaryKeyAttributes` and `EOEntity.primaryKeyForGlobalID`, as well as the API reference for `com.webobjects.eocontrol.EOClassDescription`.

Client-Side Class Property

This attribute characteristic applies only to three-tier Java Client or Cocoa client applications. It plays a vital role in the process of **business logic partitioning**. This is the process in which you choose the data that is made available to the client application.

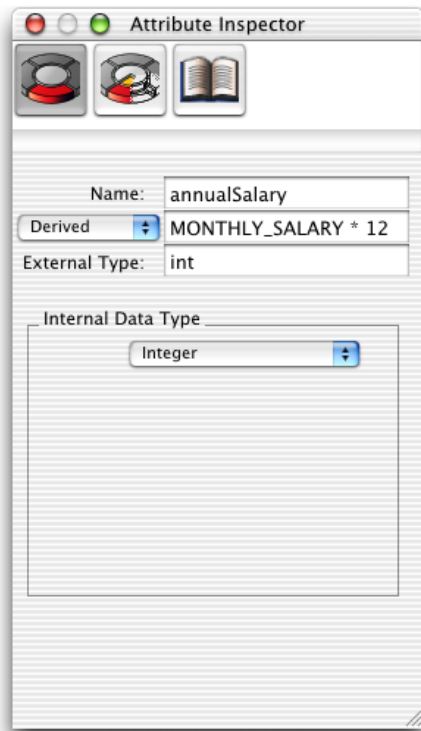
For example, a Customer entity might include an attribute called `creditCardNumber`. While this attribute is probably important to the server-side application for processing orders, it is considered sensitive data and should not be made available to client applications. To ensure that client applications don't have access to this attribute, it should not be marked as a client-side class property.

See *Inside WebObjects: Java Client Desktop Applications* for detailed information on the client-side class property characteristic, including a tutorial.

Definition (Derived Attributes)

This characteristic applies only to derived attributes. Derived attributes are usually calculated from other attributes, such as multiplying an employee's monthly salary by twelve or deriving a person's full name from first name and last name attributes. The syntax of a derived attribute definition is a SQL statement. To define an annual salary, for instance, you would multiple a `MONTHLY_SALARY` column by twelve. Figure 3-2 shows the Attribute Inspector for a derived attribute that does just this.

Figure 3-2 Derived attribute syntax



Derived attributes are effectively read-only since there is no place to write them back to. You could get the value of a derived attribute and write it back to another column but that requires another attribute. And if you need to store the value of a derived attribute, it's usually much better to perform the derivation in business logic rather than at the attribute level. (Alternatively, you could use custom read and write formats to accomplish this. See ["Read Format and Write Format"](#) (page 40)). By deriving attributes at the business-logic level, you write the code in Java, you avoid writing database-specific SQL, and you get the full benefits of enterprise objects.

One of the most important benefits is the internal update notifications that enterprise objects send and receive. In the previous example, if you change an employee's monthly salary, the derived attribute that calculates the annual salary is then incorrect. And since the attribute is derived, its value as it exists in the enterprise object is immutable. Unless the object is flushed from the access layer's snapshot and refreshed, the derived attribute is stale and inaccurate.

Derived attributes can be useful but should probably be reserved for read-only applications and can usually be replaced by values derived in business logic. Also, because derived attributes don't directly map to anything in the database, they cannot be used for locking or as primary keys.

Locking

As introduced in ["Locking"](#) (page 39), the Enterprise Object technology supports different locking strategies to deal with the problem of update conflicts. In multi-user database applications in which many users have write access, there is a possibility that multiple users may view and edit the same record concurrently. A good locking strategy can help you avoid problems if this situation arises.

The default locking strategy used by Enterprise Objects is optimistic locking. With this strategy, update conflicts aren't detected until users try to save an object's changes to the data store. At that point, Enterprise Objects checks the database row to see if it's changed since the object being edited was fetched. If the row has been changed, the save operation is rolled back and an optimistic locking exception is thrown.

Enterprise Objects determines that a database row has changed since its corresponding enterprise object was fetched using a technique called **snapshotting**. When Enterprise Objects fetches an object from the data store, it records a snapshot of the state of the corresponding database row. When changes to an object are saved to the database, the snapshot is compared with the corresponding database row to ensure that the row data hasn't changed since the object was last fetched.

Enterprise Objects creates snapshots based on the attributes that are selected for locking. The general rule is that only attributes that are guaranteed to contain a small amount of easily parseable data should be selected for locking. For example, an attribute with external type `object`, `blob`, or `clob` should never be selected for locking (except in very rare cases).

Primary Key

Primary keys are used to identify uniquely database rows and also to provide attributes for forming relationships. Each entity in your data model needs a primary key. (If your data model represents an LDAP data source via the JNDI adaptor, each entity needs an attribute named `relativeDistinguishedName`, which is roughly analogous to a primary key.) Primary keys in entities map directly to primary keys in tables.

It is generally recommended that primary keys be kept simple. It's quite common to use `int` data types as primary keys but other numerical formats also work well. Just as with locking attributes, you should avoid attributes that contain large amounts of data and you usually do not want to use binary-type attributes as primary keys. (There are some exceptions as Enterprise Objects provides facilities to compare columns that map to an Internal Data Type of `NSData`).

Depending on your application requirements, you may need to use a compound primary key—that is, a primary key composed of multiple attributes. If you must meet this requirement, Enterprise Objects provides facilities to help you with this task. You can designate a compound primary key in a model by marking multiple attributes as primary key attributes. Then, you can use the API provided in the access layer to generate custom primary keys based on these attributes. See the method description in the `com.webobjects.eoaccess` API reference for `EODatabaseContext.Delegate.databaseContextNewPrimaryKey` for more details.

You should be careful with the primary key characteristic. The primary key identified in an entity must correspond to a primary key constraint defined in the database table with which that entity is associated. So although EOModeler provides user interface to easily mark and unmark attributes as primary keys, you should not do so unless you also make a corresponding change in the database. And if you do this, you risk breaking existing relationships.

Read Format and Write Format

In addition to mapping database rows to instance variables or to an object's data, EOAttribute objects can also alter how database values are selected, inserted, and updated. This is accomplished by defining special format strings for particular attributes. These format strings allow an application to extend its reach down to the database server for certain operations. These operations are then performed by the database server, which may or may not be advantageous to your application.

Using a custom read format (for SELECT operations), you can create a kind of derived attribute without defining the attribute as derived. For example, rather than defining a derived attribute to calculate an employee's annual salary based on monthly salary multiplied by twelve, you can derive this value by setting the read format to the same SQL string you'd use were you to declare the definition for a derived attribute. The advantage of using custom read formats over derived attributes is that you can easily write the derived value back to the data source by including a complimentary write format.

The difference between attributes that are declared to be derived and attributes that are derived from custom read formats is that the latter performs an operation on itself whereas derived attributes operate on values in other attributes, often aggregating them or otherwise modifying them. So, whereas the definition of a derived attribute that calculates an annual salary based on a monthly salary would be `MONTHLY_SALARY * 12`, the read format for an attribute that does the same thing would be `%P * 12`. The former does not require a column in the database whereas the latter does.

Custom format strings can also be used for INSERT and UPDATE operations. For example, if you want to store an employee's salary in monthly terms rather than in annual terms, you would set the write format to be `%V / 12`. So, whenever the salary attribute is written back to the database, its value is divided by 12.

Read format strings use `"%P"` as the substitution character for the value that is being formatted whereas write format strings use `"%V"` as the substitution character. If, for example, you are deriving an annual salary from a column that stores salaries in monthly terms (`MONTHLY_SALARY`), the format string is `%P * 12`. So rather than sending the database server a message of `SELECT MONTHY_SALARY`, it is instead sent `SELECT MONTHY_SALARY * 12`.

You can use any legal SQL value expression in a format string and you can even use database-specific features such as functions. (A common case function is one that converts data from one type to another when it is read or written, such as converting a string to a date when writing and from a date to a string when reading). Using database-specific features affords the application more flexibility but limits its portability. You are responsible for ensuring that the SQL is well-formed and can be understood by the database server.

Using custom read and write formats is probably useful only when dealing with legacy data in which the stored data is out of sync with your current business logic. In the examples used above, the old business logic would be to store salaries based on monthly terms. The great database application you're writing uses this legacy data store but displays salaries in annual terms. To maintain the integrity of the data in the database, it's important to divide annual salary by twelve upon each commit. This transformation, however, should be transparent to the end user, so using custom read and write formats solves this problem.

Value Type

When you choose a value class for a particular attribute, you sometimes do not provide Enterprise Objects with all the information the JDBC adaptor needs to negotiate with the data source.

For example, when you specify Number as the value class for a particular attribute, you are telling Enterprise Objects to use `java.lang.Number`, which is an abstract class. This is where the value type characteristic steps in. It specifies the exact class an attribute should map to.

The possible value types for numeric attributes are as follows(note case):

- `b`—`java.lang.Byte`
- `s`—`java.lang.Short`
- `i`—`java.lang.Integer`

- l—`java.lang.Long`
- f—`java.lang.Float`
- d—`java.lang.Double`
- B—`java.math.BigDecimal`
- c—`java.lang.Boolean`

The value type also specifies which JDBC methods are used to send and retrieve the data to and from the database. These value types affect which method the `java.sql.PreparedStatement` object uses to transfer text data between the database and the JDBC adaptor. For attributes with a value class of `String`, the following value types are defined:

- <none>—uses `setString` if the text is less than the database’s advertised maximum `varchar` length and `setCharacterStream` if it is too large. If the database fails to advertise a maximum length, the default is 256 characters.
- S—uses `setString` regardless of the text’s length.
- C—uses `setCharacterString` regardless of the text’s length.
- E—converts the text into raw UTF-8 bytes and then uses `setBinaryStream` to save them in a binary-typed column in the database.
- c—tells the adaptor to generate SQL using `RTRIM` to strip off all trailing spaces.

Database columns of type `char` hold string values that are right-padded with spaces to the width of the column. String values in Enterprise Objects, however, normally do not have trailing spaces for performance and other efficiency reasons. An attribute that maps to an external type of `char` should have a value type of `c` to tell the JDBC adaptor to trim trailing spaces when fetching values that correspond to that attribute. If the value type is left blank for attributes that map to an external type of `char`, then no trimming occurs. Attributes that map to `varchar` columns are never trimmed, regardless of value type.

S is the appropriate value type for most text columns. C is good for columns that usually contain large amounts of data. c should be used when trailing spaces are not significant in a `char` column. It is not recommended to use E, except when absolutely necessary. It is the database’s responsibility to handle text encoding issues and using E usually indicates that the database is not properly configured.

For attributes with a value class of `NSTimestamp`, the following value types are defined:

- <none>—uses `getObject` on the `java.sql.ResultSet` object and `setObject` on the `java.sql.PreparedStatement` object. It assumes the database can provide a value compatible with a `java.sql.Timestamp` object.
- D—`java.util.Date` uses `setDate` and `getDate`.
- t—`java.sql.Time` uses `getTime` and `setTime`.
- T—`java.sql.Timestamp` uses `getTimestamp` and `setTimestamp`.
- M—use in place of D if using Microsoft SQL Server. Supports only `java.sql.Date`.

Prototype Attributes

To make creating models easier, EOModeler supports **prototype attributes**. These are special attributes from which other attributes derive their settings. A prototype can specify any of the characteristics you normally define for an attribute. When you create an attribute, you can associate it with one of these prototypes, and the attribute's characteristics are then set from the prototype definition.

For example, you can create a prototype attribute called `lastModified` whose value class is `Date`, whose external type is `datetime`, and which corresponds to a column named `LAST_MODIFIED`. Then, when you create other entities, you can create an attribute and associate it with this prototype and the prototype's values are copied in to the new attribute. You can then change any values in or add values to the new attribute. Any value that is inherited from the prototype that you don't override uses the value defined in the prototype.

Creating Prototype Attributes

The prototypes you can assign to attributes can come from two places:

1. An entity named `EOAdaptorNamePrototypes`, where *AdaptorName* is the name of the adaptor for your model. WebObjects 5.2 includes an adaptor for JDBC data sources and an adaptor for JNDI data sources. So you can create a prototype entity called either `EOJDBCPrototypes` or `EOJNDIPrototypes`, depending on the adaptor you use.
2. An entity named `EOPrototypes`.

To create a prototype attribute, first create a prototype entity—an entity named either `EOAdaptorNamePrototypes` or `EOPrototypes`—and add an attribute to it. Figure 3-3 shows an attribute in a prototype entity. It shows all the values that prototype attributes can define: column name, value class, external type, and value type.

Figure 3-3 A prototype entity

EOPrototypes Attributes					
Name	Column	Value Class (Java)	External Type	Value Type	Prototype
lastModified	LAST_MODIFIED	NSTimestamp	datetime	▼ D	▼

Assigning a Prototype to an Attribute

To assign a prototype attribute to an attribute, reveal the Prototype column in table mode, and select a prototype attribute from the pop-up menu. The prototype attributes that appear in the pop-up list in the Prototype column include prototype attributes defined in any entity in any model in the application's model group, which includes the current model.

Figure 3-4 shows an attribute named `lastModified` which inherits characteristics from the prototype attribute called `lastModified`. As you can see in the figure, characteristics that attributes derive from their prototype are colored differently than are other characteristics.

Figure 3-4 An attribute using a prototype

Attribute using a prototype definition

Document Attributes					
Name	Column	Value Class (Java)	External Type	Value Type	Prototype
lastModified	LAST_MODIFIED	NSTimestamp	datetime	▼ D	lastModified ▼
title	TITLE	String	char	▼	▼

When you use prototype attributes, in some cases you want to derive only some of the values from the prototype. To do this, just set the characteristic of the attribute to the value you want. The rest of the derived characteristics still resolve to the values set in the prototype. The prototype selected in the Prototypes column then appears with an asterisk. Figure 3-5 shows an attribute that uses only part of a prototype definition.

Figure 3-5 An attribute using part of a prototype

Attribute using part of a prototype definition

Document Attributes					
Name	Column	Value Class (Java)	External Type	Value Type	Prototype
lastModified	LAST_MODIFIED	NSTimestamp	datetime	▼ t	lastModified * ▼
title	TITLE	String	char	▼	▼

Flattened Attributes

A flattened attribute is a special kind of attribute that you effectively add from one entity to another by traversing a relationship. When you form a to-one relationship between two entities (such as Person and PersonPhoto), you can add attributes from the destination table to the source table. For example, you can add a `personPhoto` attribute to the Person entity. This is called “flattening” an attribute and is equivalent to creating a joined column—it allows you to create objects that extend across tables.

When Should You Flatten Attributes?

Flattening attributes is just another way to conceptually “add” an attribute from one entity to another. A generally better approach is to traverse the object graph directly through relationships. Enterprise Objects makes this easy by supporting the notion of key paths.

The difference between flattening attributes and traversing the object graph (either programmatically or by using key paths) is that the values of flattened attributes are tied to the database rather than the object graph. If an enterprise object in the object graph changes, a flattened attribute can quickly get out of sync.

For example, suppose you flatten a `departmentName` attribute into an Employee object. If a user then changes an employee’s department reference to a different department or changes the name of the department itself, the flattened attribute won’t reflect the change until the changes in the object graph are committed to the database and the data is refetched (this is because flattened attributes are derived attributes—see “[Definition \(Derived Attributes\)](#)” (page 38) for more details). However, if you’re using key paths in this scenario, users see changes to data as soon as they happen in the object graph. This ensures that your application’s view of the data remains internally consistent.

Therefore, you should use flattened attributes only in the following cases:

- If you want to combine multiple tables joined by a one-to-one relationship to form a logical unit. For example, you might have employee data that's spread across multiple tables such as ADDRESS, BENEFITS, and so on. If you have no need to access these tables individually (that is, if you'd never need to create an Address object since the address data is always subsumed in the Employee object), then it makes sense to flatten attributes from those entities into the Employee entity.
- If your application is read-only.
- If you're using vertical inheritance mapping. See ["Vertical Mapping"](#) (page 69).

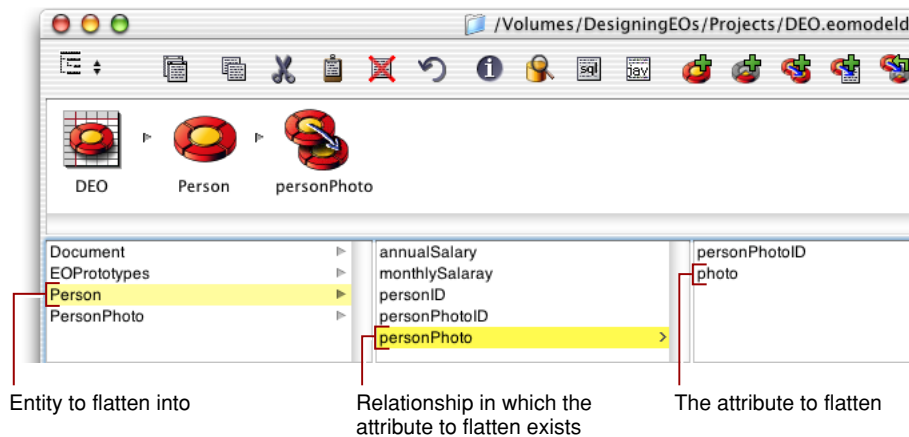
Flattening an Attribute

To flatten an attribute:

1. In browser mode, select the entity in which you want the flattened attribute to appear and select the relationship whose destination entity holds the attribute you want to flatten.

For example, in the Real Estate model, to flatten the `photo` attribute from the `PersonPhoto` entity into the `Person` entity, select the `Person` entity and select the `personPhoto` relationship, as shown in Figure 3-6.

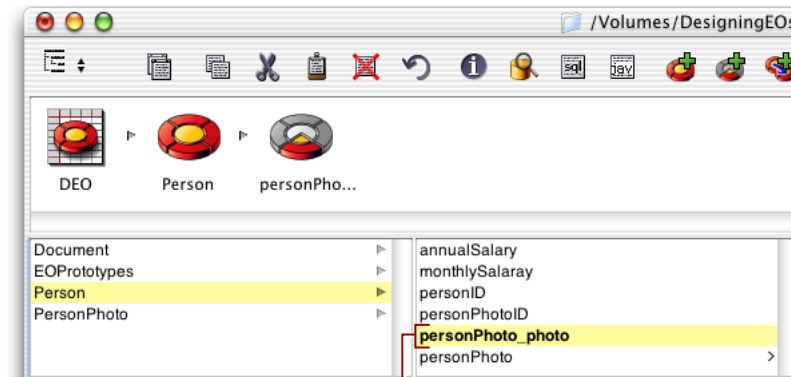
Figure 3-6 Selecting the relationship in which the attribute to flatten exists



2. Select the attribute in the destination entity (`photo`) for which you want to create the flattened attribute and choose Flatten Property from the Property menu.

The flattened attribute appears in the list of properties for the Person entity as `personPhoto_photo`, as shown in Figure 3-7. The format of the name reflects the traversal path: the attribute `photo` is added to the Person entity by traversing the `personPhoto` relationship.

Figure 3-7 An attribute flattened



The flattened attribute

If you select the flattened attribute and display the Attribute Inspector, you'll see that the attribute is considered derived and its definition is a key path, as shown in Figure 3-8. Just as with other types of attributes, you can edit the flattened attribute's name in the inspector.

Figure 3-8 A flattened attribute in the Attribute Inspector



Working With Relationships

This chapter describes how to create and configure relationships in EOModeler. It also provides an introduction to relationships. It is organized in the following sections:

- [“About Relationships”](#) (page 47) introduces the concept of relationships as defined by the entity-relational paradigm. It also discusses basic principles of relationships such as cardinality, joins, and relationship keys.
- [“Creating Relationships”](#) (page 50) describes how to use EOModeler to create relationships.
- [“Tips for Specifying Relationships”](#) (page 53) provides some general design patterns when creating relationships.
- [“Adding Referential Integrity Rules”](#) (page 54) discusses the various referential integrity rules supported by Enterprise Objects, such as the delete rules and optionality.
- [“Flattened Relationships”](#) (page 55) describes what flattened relationships are and how to form them in EOModeler.
- [“Modeling Many-to-Many Relationships”](#) (page 57) discusses how to build many-to-many relationships using EOModeler.

About Relationships

Relational databases derive much of their value from the relationships between the tables they store. Likewise, the Enterprise Object technology includes infrastructure that brings relationship data to life in data-driven applications.

A relationship expresses the affinity between tables in a data source. In the most simple case, a relationship expresses a meaningful connection between two tables in a data source. You can also think of relationships as cross-references much like entries in a book’s index. A single index entry can cross-reference one or more other index entries so that there is a relationship between index entries.

For example, a `Person` table could be related to a `PersonPhoto` table by a relationship called `toPhoto`. In relationship lingo, the `Person` table is referred to as the source table or source entity that contains source records. The `PersonPhoto` table is referred to as the destination table or destination entity that contains destination records.

Directionality

Relationships are unidirectional, which means that the path that leads from the source to the destination can’t be traveled in the opposite direction—you can’t use a relationship to go from the destination to the source. So, although you can use a `toPhoto` relationship to find the photo for a particular person, you can’t use this relationship in reverse to get a person’s name.

Unidirectionality is enforced by the way a relationship is resolved. Resolving a relationship means finding the correct destination record or records given a specific source record.

Bidirectional relationships—in which you can look up records in either direction—can be created by adding a separate return-trip, or inverse, relationship. But there is no concept of a single relationship that is bidirectional.

Cardinality

Every relationship has a **cardinality**. The cardinality defines how many destination records can potentially resolve the relationship. In relational database systems, there are generally two cardinalities:

- to-one relationship—for each source record, there is exactly one corresponding destination record
- to-many relationship—for each source record, there may be zero, one, or more corresponding destination records

An `employeeDepartment` relationship is an example of a to-one relationship: An employee can be associated with only one department in a company. An `Employee` entity might also be associated with a `Project` entity. In this case, there would be a to-many relationship from `Employee` to `Project` called `projects` since an `Employee` can have many projects.

Relationship Keys

The construction of a relationship requires that you designate at least one attribute in each entity as a **relationship key**. Relationship keys are necessary so a relationship can be resolved. For example, the `toPhoto` relationship which relates a `Person` entity to a `PersonPhoto` entity, uses two relationship keys: `personPhotoID`, the source key in the `Person` entity, and `personPhotoID`, the destination key in the `PersonPhoto` entity.

Figure 4-1 shows the `PERSON` table's columns.

Figure 4-1 Foreign key for `PersonPhoto` in `Person` table

PERSON_ID	MONTHLY_SALARY	PERSON_PHOTO_ID
2	3000	1
3	4400	2
4	3750	3

When Enterprise Objects resolves this relationship, it creates a join table by looking up the `PERSON_PHOTO_ID` key in the `PERSON_PHOTO` table. In Figure 4-1, the row with `PERSON_ID = 2` has a value of 1 in the `PERSON_PHOTO_ID` column. The relationship specifies that the `PERSON_PHOTO_ID` column in the `PERSON` table resolves to the `PERSON_PHOTO_ID` column in the `PERSON_PHOTO` table. As shown in Figure 4-2, the row with `PERSON_PHOTO_ID = 1` in the `PERSON_PHOTO` table holds binary data that represents a photo.

Figure 4-2 `PersonPhoto` primary key in `PersonPhoto` table

PERSON_PHOTO_ID	PHOTO
1	asdf
2	af32q43z
3	ghngdh2423g

There are some general guidelines when choosing which attributes to use as relationship keys. In to-one relationships, the destination key must be a primary key in the destination entity. In to-many relationships, the destination key is usually a foreign key in the destination entity (which is most often a copy of the source entity's primary key). The source key or foreign key should emulate the destination key in that the data types must be the same and the names should be similar.

So, in the previous example, `PERSON_PHOTO_ID` is the primary key for the `PERSON_PHOTO` table and it is a column of type `int`. In the `PERSON` table, `PERSON_PHOTO_ID` is a foreign key that is of the same type and name as the primary key it maps to in the `PERSON_PHOTO` table.

When you use relationship keys to express an affiliation between two entities, keep in mind these general rules:

- For to-one relationships, the source attribute is a foreign key in the source entity while the destination key is the primary key of the destination entity.
- For to-many relationships, the source attribute is the primary key in the source entity (but it can also be a foreign key in the source entity) while the destination key is a foreign key of the destination entity.

If you have consistency checking enabled in EOModeler, it warns you if any to-one relationships in your model have destination keys that are not primary keys.

Reflexive Relationships

A unique kind of relationship is the **reflexive relationship**—a relationship that shares the same source and destination entity. Reflexive relationships are important when modeling data in which an instance of an entity points to another instance of the same entity.

For example, to show who a given person reports to, you could create a separate manager entity. It would be easier, however, to just create a reflexive relationship. The `managerID` attribute is the relationship's source key whereas `employeeID` is the relationship's destination key. Where a person's `managerID` is the `employeeID` of another employee object, the first employee reports to the second. If an employee doesn't have a manager, the value for the `managerID` attribute is `null` in that employee's record.

Figure 4-3 shows this relationship as it exists in the Employee table. The row with `NAME = Brent` references the row with `NAME = K. Boss` since the manager relationship is defined with `MANAGER_ID` as the source key and `EMPLOYEE_ID` as the destination key.

Figure 4-3 Reflexive relationship table

EMPLOYEE_ID	MANAGER_ID	NAME
1		K. Boss
2	1	Brent

← This record references this record —

Reflexive relationships can represent arbitrarily deep recursions. So, in the model above, a person can report to another person who reports to yet another person, and so on. This could go on until a person's `managerID` attribute is `null`, which denotes that person reports to no one.

Owns Destination and Propagate Primary Key

The Owns Destination option lets you specify whether the relationship's source owns its destination objects. When a source object owns its destination object, for example, as when an Agent object owns its Customer objects, when a destination object (Customer) is removed from the relationship, it is also removed from the data source. Ownership implies that an owned object cannot exist without its owner.

The Propagate Primary Key option lets you specify that the primary key of the source entity should be propagated to newly inserted objects in the destination of the relationship. That is, when inserting objects that are the destination of the relationship, this option suppresses primary key generation for the destination entity and instead uses the source object's primary key as the primary key for the newly inserted destination object.

This option is used for an owning relationship where the owned object has the same primary key as the source. Propagating primary key confers a performance improvement as it doesn't require the generation of a primary key for the destination entity. Primary key propagation is also commonly used to generate primary keys for join tables in many-to-many relationships.

Creating Relationships

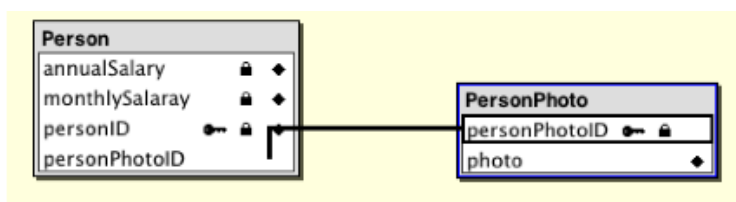
If the data source on which your model is based includes foreign key definitions, these definitions are automatically expressed in your model when you create a model from an existing data source with the New Model Wizard. But if you are creating the schema within EOModeler, you need to define relationships in the model editor.

EOModeler provides two mechanisms for forming relationships. You can form them in the model editor's diagram view or in the Relationship Inspector. Using the diagram view is the quickest way to create new relationships, but using the Relationship Inspector gives you access to more relationship characteristics. Each mechanism is discussed in the following sections.

Forming Relationships in the Diagram View

To create a relationship in diagram view, Control-drag from a source attribute to a destination attribute, as shown in Figure 5-4.

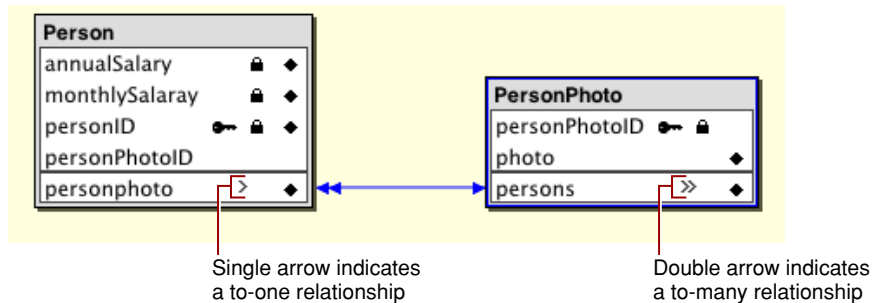
Figure 4-4 Control-drag from source key to destination key to form a relationship



The cardinality of the relationship is determined by the primary key characteristic of the destination attribute. If the destination attribute is the entity's primary key, the relationship is modeled as a to-one relationship. If the destination attribute is a foreign key, the relationship is modeled as a to-many relationship.

Control-dragging to form a relationship actually creates two relationships: one in the source attribute's entity and an inverse relationship in the destination attribute's entity. In Figure 4-5, Control-dragging from `Person.personPhotoID` (a foreign key) to `PersonPhoto.personPhotoID` (a primary key) creates a to-one relationship from the `Person` entity to the `PersonPhoto` entity and a to-many relationship from the `PersonPhoto` entity to the `Person` entity.

Figure 4-5 Control-dragging also creates an inverse relationship



In Figure 4-5, the single line indicating the relationships between the `Person` and `PersonPhoto` entities should not be mistaken for a “bidirectional” relationship, which is not possible in the object-relational model. It is actually two relationships but when you create a relationship in diagram view, it appears as a single line.

For the `personPhoto` relationship, it doesn't make sense for the inverse relationship (`persons` in `PersonPhoto`) to be a to-many relationship. You can make it a to-one relationship in the Relationship Inspector as well as set other characteristics of the relationship. The Relationship Inspector is described in detail in “[Forming Relationships in the Inspector](#)” (page 51) and “[Forming Relationships Across Models and Data Sources](#)” (page 52).

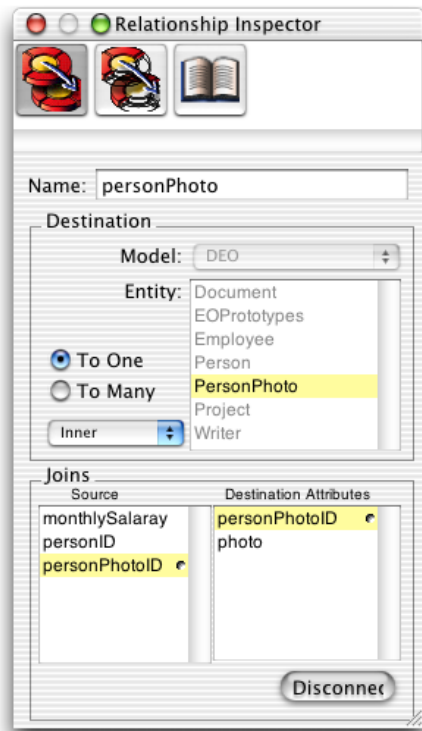
Forming Relationships in the Inspector

Creating relationships in the Relationship Inspector is a more manual process than creating one in the diagram view. The inspector provides only the ability to configure a relationship that already exists. So before you can edit a relationship in the Relationship Inspector, you must add a relationship to an entity.

Assuming that the entities in the previous example exist (`Person` and `PersonPhoto`), you can form a relationship between them by selecting an entity (`Person`) and then choosing `Add Relationship` from the Property menu. Then, select the new relationship in the tree view by clicking the plus sign next to an entity and open the Relationship Inspector by choosing `Inspector` from the Tools menu.

Now you can use the Relationship Inspector to configure the relationship. Follow these steps and refer to Figure 4-6 for clarity:

1. In the Relationship Inspector, select the destination entity (`PersonPhoto`) from the Entity list in the Destination box.
2. Select the source attribute (`personPhotoID`) in the Source Attributes list.
3. Select the destination attribute (`personPhotoID`) in the Destination Attributes list.
4. Make sure the relationship has the proper cardinality, `To One` in this case.
5. Click `Connect`.

Figure 4-6 Using the Relationship Inspector to build a relationship

EModeler assigns the relationship a default name based on the name of the destination entity and the cardinality of the relationship. You can edit this name using the Relationship Inspector or in table mode.

The source and destination attributes you chose are based on general rules for relationships, which are described in “[Relationship Keys](#)” (page 48).

Forming Relationships Across Models and Data Sources

The entities in one model can have relationships to the entities in another model. You can form such relationships even if the models map to different databases and different database servers.

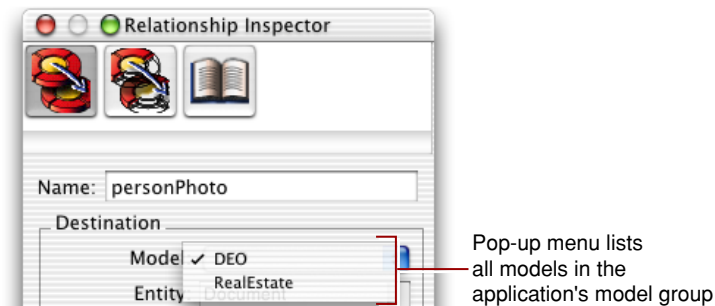
When you add a model to a project, it becomes part of a model group. Every Enterprise Objects application includes a default model group, even if the project contains only one model. Each model you add to a project automatically becomes part of the project’s model group. The only regulation between multiple models in a model group is that entity names must be unique.

To form a relationship from one model to another, use the Relationship Inspector as follows:

1. Add a relationship to the entity you want to use as the source of the relationship.

2. In the Relationship Inspector, use the Model pop-up menu to choose the model that contains the entity you want to use as the relationship's destination. This menu displays all the models in the application's model group. Figure 4-7 shows two models in an application's model group, RealEstate and DEO.

Figure 4-7 Multiple models to choose from



3. Specify the relationship as you normally would.

Note: You can't flatten properties across databases, nor can you map inheritance hierarchies across databases (though you can do both of these things across models that map to the same database).

Tips for Specifying Relationships

The following tips are useful to keep in mind as you add relationships to your model:

- The relationships you define in a model must reflect corresponding implementations in the data source, as well as the features supported by the adaptor your model uses. Enterprise Objects doesn't know, for example, if your adaptor supports left outer joins, so you need to be careful with regard to the characteristics you set for relationships.
- Use the diagram view to quickly create pairs of inverse relationships by Control-dragging between source and destination attributes.
- Use the Relationship Inspector to specify information about a relationship, such as whether it's to-one or to-many, its semantics (join type), the destination model, and the destination entity in the destination model.
- Relationships can be compound, meaning that they can consist of multiple pairs of connected attributes. You can specify additional pairs of attributes only in the Relationship Inspector. Simply select a second source attribute and a second destination attribute and click Connect a second time.
- A to-one relationship from one foreign key to a primary key must always have exactly one row in the destination entity—if this isn't guaranteed to be the case, use a to-many relationship. This rule doesn't apply to a foreign key to primary key relationship where a `null` value for the foreign key in the source row indicates that no row exists in the destination.
- To-one relationships must join on the complete primary key of the destination entity.

Adding Referential Integrity Rules

You can use the Advanced Relationship Inspector to add referential integrity rules for relationships. These rules specify relationship characteristics such as optionality and delete rule. The referential integrity rules you specify are not written back to the data source: When Enterprise Objects creates objects for relationships, the objects include the referential integrity rules you specify in the model.

Note: The referential integrity rules you specify apply only to the object graph managed by Enterprise Objects. Therefore, if your data source also specifies referential integrity rules, you are responsible for avoiding or managing any conflicts between the two sets of rules.

Optionality

The Optionality section lets you specify whether a relationship is optional or mandatory. For example, you could require that all Document objects have a related Writer object but not require that all Document objects have a related Illustrator object. When you attempt to save an object that has a mandatory relationship that is not set (so the relationship is `null`), Enterprise Objects refuses the save and displays an error message stating that the object being saved has a mandatory relationship that must be set.

Delete Rule

The options in the Delete Rule section specify what to do when the source object of a relationship is deleted. There are four options:

- **Nullify** disassociates all destination objects from the source object by removing references to them. So, when an Agent object is deleted, its related Customer objects are not deleted but the Customer objects' references to Agent are nullified (the entry in the join table is set to `null`).
- **Cascade** deletes all objects that are the destination of a relationship whose source is deleted. So, when an Agent object is deleted, all of its related Customer objects are also deleted.
- **Deny** refuses the deletion if a source object has any destination objects. So, if an Agent object has any Customer objects, deleting the Agent object is denied. In order for the deletion of the Agent object to succeed, its destination objects (Customer objects) must either be deleted or changed to something other than destination objects of the Agent object.
- **No Action** deletes the destination object but does not remove any back references to the source object. So, if a Customer object is deleted, its reference to its Agent object is not removed. Using this option may result in dangling references in the data source.

Flattened Relationships

Just as you can flatten attributes (see “[Flattened Attributes](#)” (page 44)), you can also flatten relationships. Flattening a relationship gives a source entity access to relationships that a destination has with other entities. It’s equivalent to performing a multitable join. Note that flattening either an attribute or a relationship can result in degraded performance when the destination objects are accessed, since traversing multiple tables makes fetches slower.

When Should You Flatten Relationships?

As discussed in “[When Should You Flatten Attributes?](#)” (page 44), flattening is a technique you should use only under certain conditions. Instead of flattening an attribute or a relationship, you can instead directly traverse the object graph, either programmatically or by using key paths. This ensures that your application maintains an internally consistent view of its data.

There is one scenario in which you might want to use a flattened relationship: If you’re modeling a many-to-many relationship and you want to perform a multitable hop to access a table that lies on the other side of an intermediate table.

For example, in the Real Estate database, the Suggestion table acts as an intermediate table between Customer and Listing. It’s highly unlikely that you would ever need to fetch instances of Suggestion into your application. In this situation, it makes sense to specify a relationship between Customer and Suggestion and flatten that relationship to give Customer access to the Listing table.

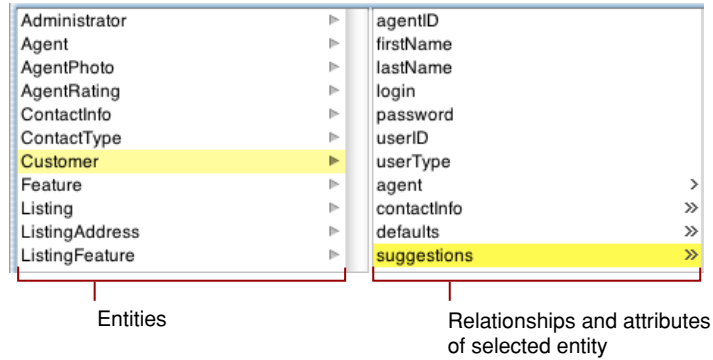
Flattening a Relationship

Follow these steps to flatten a relationship:

1. Add a relationship from one entity (entity_1) to a second entity (entity_2). For example, add a to-many relationship called `suggestions` from Customer to Suggestion.
2. Add a relationship from entity_2 (Suggestion) to a third entity (entity_3, Listing). For example, add a to-many relationship called `listing` from Suggestion to Listing.

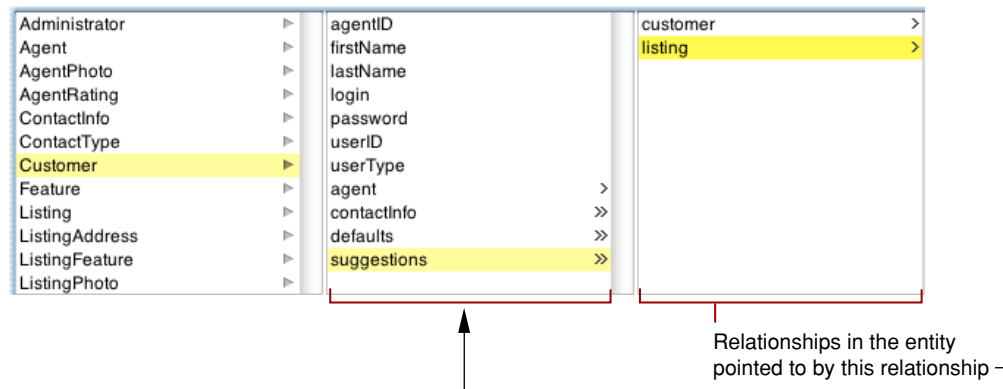
3. In browser mode, from entity_1 (Customer), select the relationship to entity_2 (suggestions) to display its attributes and relationships.

Figure 4-8 Select the relationship that contains the relationship to flatten



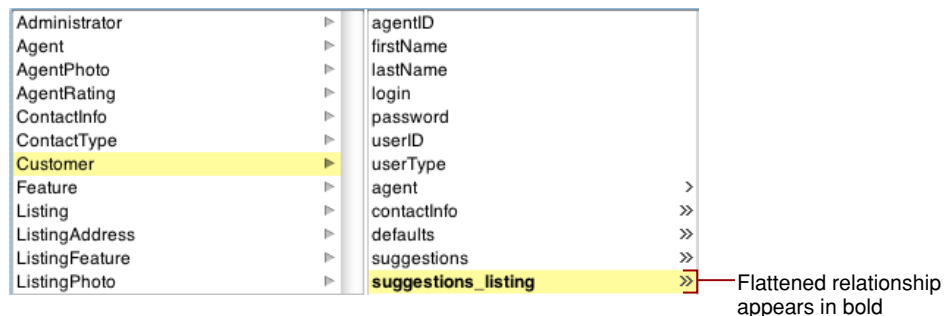
4. In the list of attributes and relationships for entity_2 (Suggestion), select the relationship (listing) you want to flatten.

Figure 4-9 Select the relationship to flatten



5. Choose Flatten Property from the Property menu. The flattened relationship should appear in bold typeface as shown in Figure 4-10; the name is derived from the relationship key path.

Figure 4-10 A flattened relationship displayed in browser mode



The flattened relationship (in this example, `suggestions_listing`) appears in the list of properties for Customer.

Modeling Many-to-Many Relationships

Modeling a many-to-many relationship between objects is simple: Each object manages a collection of the other kind. For example, consider the many-to-many relationship between employees and projects. To think of this relationship in objects, an Employee object has an array of Project objects representing all of the projects the employee works on; and a Project object has an array of Employee objects representing the people working on the project.

To model a many-to-many relationship in a database, you have to create an intermediate table (also known as a correlation or join table). For example, the database for employees and projects might have `EMPLOYEE`, `PROJECT`, and `EMPLOYEE_PROJECT` tables, where `EMPLOYEE_PROJECT` is the correlation table.

Given the relational database representation of a many-to-many relationship, how do you get the object model you want? You don't want to see evidence of the correlation table in your object model, and you don't want to write code to maintain database correlation rows. With Enterprise Objects, fortunately, you don't have to. You can simply use flattened relationships to hide correlation tables.

A model with the following features has the effect of hiding the `EMPLOYEE_PROJECT` correlation table from its object model:

- Employee and Project entities whose to-many relationships to the EmployeeProject entity are not class properties. These to-many relationships (named `projectEmployees` in this example) are never instantiated or used at the application level.
- The flattened relationships `projects` and `employees` in Employee and Project, respectively, are class properties.

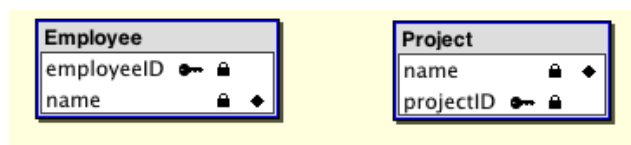
Consequently, EmployeeProject enterprise objects are never created, Employee objects have an array of related Projects, and Project objects have an array of related Employees. Furthermore, Enterprise Objects automatically manages rows in the `EMPLOYEE_PROJECT` correlation table.

Still, creating a model with the parameters described in this section would take quite a bit of work and would be error prone. Fortunately, EOModeler does all the work for you.

Follow these steps to create a many-to-many relationship between two entities:

1. Switch to diagram view.
2. Select the entities you want to join in a many-to-many relationship.

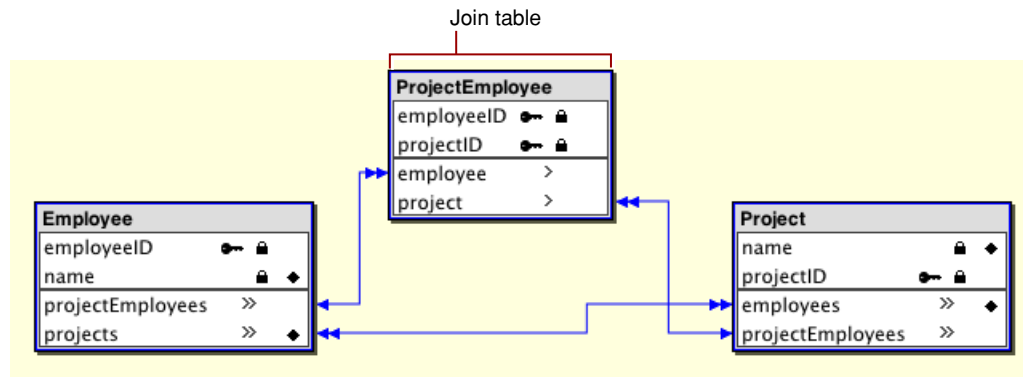
Figure 4-11 Two entities before joining in a many-to-many relationship



3. Choose Join in Many-to-Many from the Property menu.

This creates a join table between the two entities, adds flattened relationships in the two entities, and sets the class property characteristic for the new relationship as described in this section. The two entities in Figure 4-11 when joined in a many-to-many relationship appear in the diagram view as shown in Figure 4-12.

Figure 4-12 Two entities after being joined in a many-to-many relationship



Working With Entities

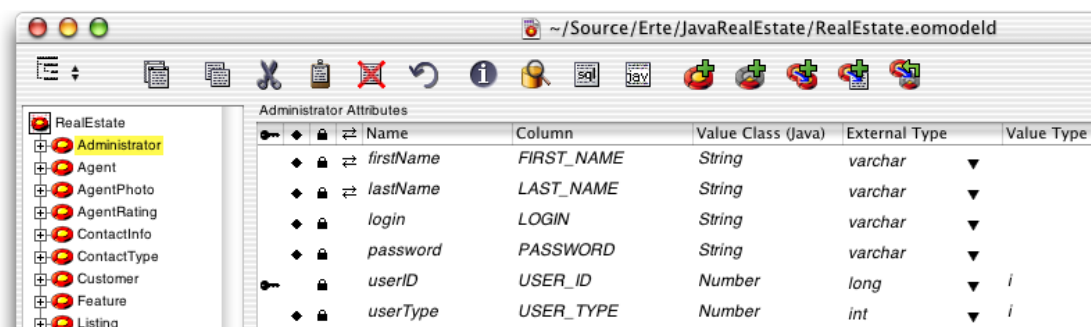
This chapter teaches you how to work with entities in EOModeler. It is divided into the following sections:

- “Entity Characteristics” (page 59) describes the characteristics of entities.
- “Advanced Entity Inspector” (page 60) describes the advanced characteristics you can assign to entities.
- “Shared Objects Inspector” (page 62) describes how to configure entities to be fetched into the shared editing context.
- “Stored Procedure Inspector” (page 63) describes how to configure entities to invoke stored procedures when certain actions occur.

Entity Characteristics

To display a model’s entities in table mode, select the model object in the tree view. Doing this displays the main characteristics of each entity such as class name and table name. To display the attributes of a particular entity, select an entity in the tree view. Figure 5-1 shows an entity called Administrator selected in the tree view and its attributes displayed in the table.

Figure 5-1 The Administrator entity selected in the tree view



The screenshot shows the EOModeler application window with the title bar indicating the file path: ~/Source/Erte/JavaRealEstate/RealEstate.eomodeld. On the left, a tree view displays a hierarchy of entities under 'RealEstate', with 'Administrator' selected. On the right, a table titled 'Administrator Attributes' displays the attributes of the selected entity. The table has six columns: Name, Column, Value Class (Java), External Type, and Value Type. The attributes listed are firstName, lastName, login, password, userID, and userType.

Name	Column	Value Class (Java)	External Type	Value Type
firstName	FIRST_NAME	String	varchar	
lastName	LAST_NAME	String	varchar	
login	LOGIN	String	varchar	
password	PASSWORD	String	varchar	
userID	USER_ID	Number	long	i
userType	USER_TYPE	Number	int	i

Each table column corresponds to a single characteristic of an entity such as its name or the name of its database table. By default, the columns included in the table—Name, Table, and Class Name—only represent a subset of the possible characteristics you can set for a given entity. Figure 5-1 shows some of the possible additional columns such as Value Type and Client-Side Class Property. To add columns for additional characteristics, you use the Add Column pop-up menu in the lower left-corner of the table. To remove a column, select it and press the Delete key.

Table 5-1 describes the characteristics you can set for an entity in the model editor.

Table 5-1 Entity characteristics

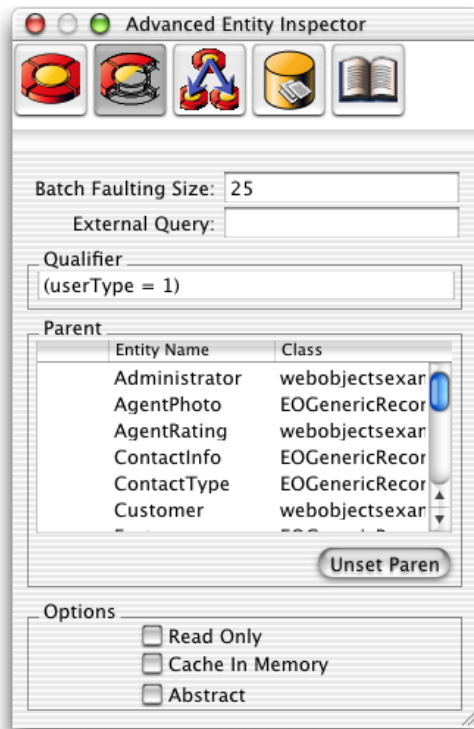
Characteristic	Description
Class Name	The name of the class that corresponds to the entity. If you don't define a custom enterprise object class for an entity, the class name defaults to EOGenericRecord. You should use a fully qualified name but this isn't strictly required.
Client-Side Class Name	The name of the class that corresponds to the entity in the client side of a three-tier WebObjects Java Client application. If you don't define a client-side class, Enterprise Objects looks for a class in the client with the same name as the server-side enterprise object class. If no such class exists on the client, it uses EOGenericRecord. You should use a fully qualified name but this isn't strictly required.
External Query	Any valid SQL statement that you want executed when unqualified fetches are performed on the entity.
Name	The name your application uses for the entity. By default, EOModeler supplies a name based on the name of the corresponding table in the data source.
Open Entity	Adds a column with an icon which you can double-click to display an entity's attributes.
Parent	Specifies an entity's parent when using inheritance.
Qualifier	Specifies a restricting qualifier that is added to every fetch specification performed on the entity. Used when modeling inheritance hierarchies.
Read Only	Specifies if the entity is read-only.
Table	The name of the table in the data source that corresponds to the entity.

Other characteristics of entities can be set in the entity inspectors.

Advanced Entity Inspector

The Advanced Entity Inspector lets you set more complex behavior for entities. To display it, click the Advanced Entity Inspector button at the top of the window. It appears as shown in Figure 5-2.

Figure 5-2 The Advanced Entity Inspector



The following list describes the characteristics that can be set in the Advanced Entity Inspector:

- **Batch Faulting Size** lets you specify the number of faults that should be triggered when you first access an object of this type that is the destination of a to-many relationship. By providing a number in this field, you specify that number of faults of the same entity should be fetched from the data source along with the first fault. This improves performance by minimizing round trips to the data source.
- **External Query** lets you specify any SQL statement to execute when Enterprise Objects performs an unqualified fetch on the entity. The columns selected by this SQL statement must be in alphabetical order by internal name and must match in number and type with the class properties specified for the entity.
- **Qualifier** is used to specify a restricting qualifier. A restricting qualifier maps an entity to a subset of rows in a table. When you add a restricting qualifier to an entity, it invokes a fetch for that entity to retrieve objects only of the type specified by the restricting qualifier. See [“Implementing Single-Table Mapping in a Model”](#) (page 77) for more information on restricting qualifiers.
- **Parent** is used to specify a parent entity for the current entity. This field is used to model inheritance. See [“Modeling Inheritance”](#) (page 67) for more details on this topic.
- **Read Only** specifies whether the data that’s represented by the entity can be altered by your application. This does not lock objects at the database level but rather works at a higher level (in the `com.weboobjects.eoaccess.EODatabaseContext` object) so that if you try to save changes to data that’s marked as read only, Enterprise Objects refuses the save and throws an exception.

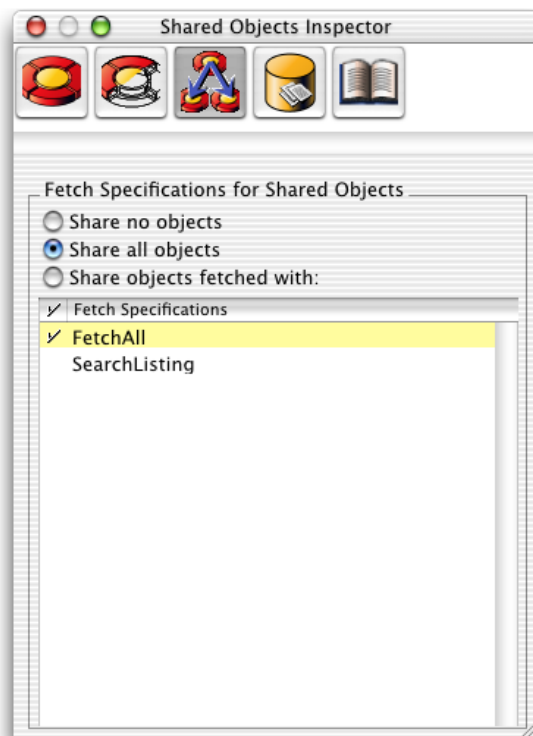
- **Cache in Memory** specifies that when one record in a table is fetched, the entire table is fetched into memory. Caching an entity's objects allows Enterprise Objects to evaluate queries in memory, thereby avoiding round trips to the data source. This is most useful for read-only entities where there is no danger of the cached data getting out of sync with the data in the data source.
- **Abstract** lets you specify whether the entity is abstract. An abstract entity is one for which no objects are ever instantiated. For example, in the Real Estate database, the User entity is abstract and is never instantiated, whereas entities that inherit from it, such as Agent and Customer, are concrete classes that are instantiated. Like the Parent field, this option is used when modeling inheritance.

Shared Objects Inspector

The Shared Objects Inspector lets you configure a feature of Enterprise Objects called the shared editing context. This is a special kind of editing context, a subclass of `EOEditingContext`: `com.webobjects.eocontrol.EOSharedEditingContext`. The shared editing context is a mechanism that allows `EOEditingContext` objects to share enterprise objects. Proper use of it can reduce redundant data in your application and limit the number of fetches to the data store an application performs.

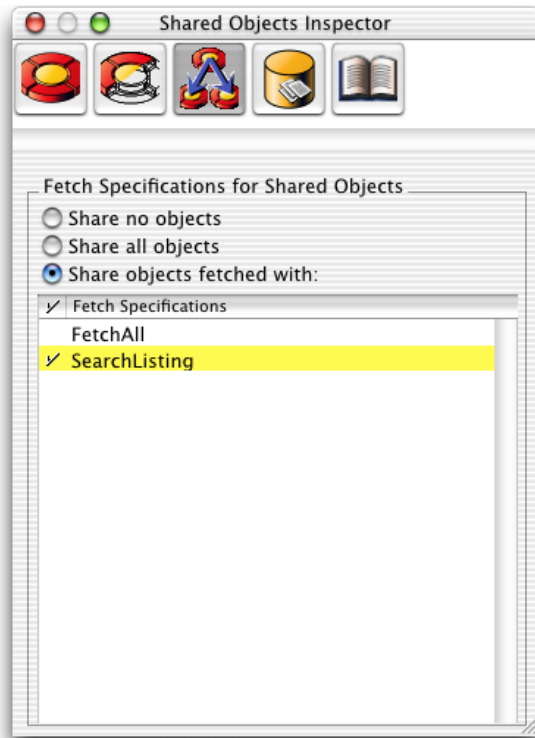
The Shared Objects Inspector provides a simple interface to define, on a per-entity basis, the objects that are fetched into the shared editing context. By selecting the Share All Objects option, all rows of data for a particular entity are fetched into the shared editing context. When you select this option, a fetch specification is added to the entity called `FetchAll`. This fetch specification performs an unqualified fetch on the entity in which it is defined. Figure 5-3 shows the Shared Objects Inspector configured with this option.

Figure 5-3 Shared Objects Inspector configured to perform an unqualified fetch



Using the Shared Objects Inspector, you can also share objects that are fetched with a fetch specification defined in that entity. Figure 6-4 shows the inspector configured to put only the objects fetched with the SearchListings fetch specification into the shared editing context.

Figure 5-4 Shared Objects Inspector configured to shared objects from a qualified fetch



Shared editing context objects are created when an EODatabaseContext object is instantiated. By default, each application instance has a single instance of EODatabaseContext. See the API reference documentation for EOSharedEditingContext for more information.

Stored Procedure Inspector

You can access the Stored Procedure Inspector by clicking the third button in the Entity Inspector. You use the Stored Procedure Inspector to specify stored procedures that should be executed when a particular database operation (such as insert or delete) occurs. In the field associated with the database operation for which you want the stored procedure to execute, you enter the stored procedure's name. The stored procedures you specify must correlate to the stored procedures in your model.

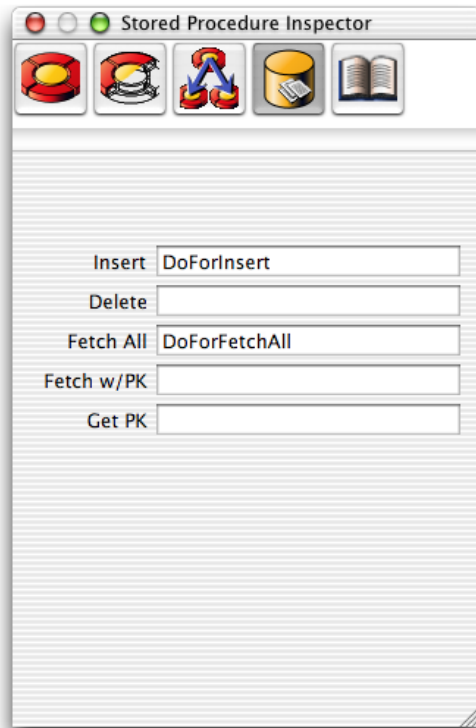
You can set up an EOModel so that Enterprise Objects automatically invokes a stored procedure for these operations on an entity:

- **Insert** to insert a new object into an entity
- **Delete** to delete an object from an entity
- **Fetch All** to fetch all objects in an entity

- **Fetch w/ PK** to fetch the object in an entity with a particular primary key
- **Get PK** to generate a new primary key for an entity

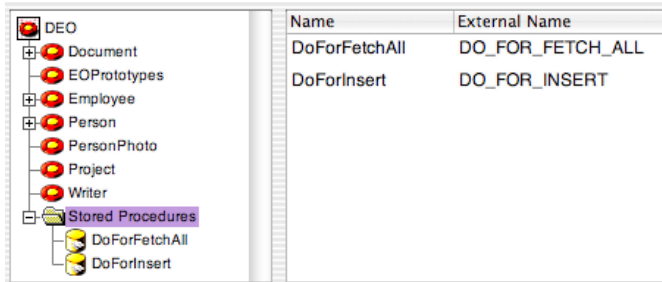
You associate a stored procedure with each of these operations by entering the stored procedure's name in the inspector, as shown in Figure 5-5.

Figure 5-5 Stored Procedure Inspector



The stored procedures you enter in the Stored Procedure Inspector must correspond to a stored procedure in the model. If you created the model from an existing data source and chose the Ask About Stored Procedures option in the wizard, stored procedures are already added to the model. If this is not the case, however, you can add stored procedures to the model using the Add Stored Procedure command from the Property menu.

When you add a stored procedure to a model, you assign it a name to use within Enterprise Objects and you associate it with the stored procedure in the data source by supplying its external name. Figure 5-6 shows two stored procedures in a model, DoForFetchAll, which is associated with the DO_FOR_FETCH_ALL stored procedure in the data source and DoForInsert, which is associated with the DO_FOR_INSERT stored procedure in the data source.

Figure 5-6 Map stored procedure in model to procedure in data source

Finally, some stored procedures take arguments that you can also define in EOModeler. To do this, select a stored procedure in the tree view, which displays its arguments in the table. You can add arguments to a stored procedure by choosing Add Argument from the Property menu. Figure 5-7 shows arguments in a stored procedure.

Figure 5-7 Stored procedure arguments

0	Name	Direction	External Type	Value Class (Java)
✓	title	In	char	String
✓	writer	In	char	String

In order for Enterprise Objects to automatically invoke a stored procedure for these operations, you must adhere to the requirements for each type of operation.

For each of the operations, if the stored procedure associated with an operation returns a value, Enterprise Objects ignores the return value.

For Fetch All operations, the stored procedure must not take any arguments and it should return a result set for all the objects in the corresponding entity. The rows in the result set must contain values for all the columns Enterprise Objects would fetch if it were not using the stored procedure, and it must return them in alphabetical order.

That is, the stored procedure should return values for primary keys, foreign keys used in class property joins, class properties, and attributes used for locking. These values must be returned in alphabetical order with regard to the attributes with which they are associated. For example, consider a Listing entity that has the attributes `listingID`, `bedrooms`, and `sellingPrice`. A stored procedure that fetches all the Listing objects should return the value for a listing's number of bedrooms, then its `listingID`, and then its selling price.

For Fetch w/ PK operations, the stored procedure must take an "in" argument for each of the entity's primary key attributes (most entities have a single primary key attribute). The argument names must match the names of the entity's primary key attributes. For example, a Listing entity has a single primary key attribute named `listingID`, so the stored procedures argument as defined in the model must also be `listingID`.

A Fetch w/ PK operation stored procedure should return a result set containing the row that matches the primary key passed in by the argument. The row must be in the same form as rows returned by the Fetch All operation.

For Insert operations, the stored procedure must take an "in" argument for each of the corresponding entity's attributes. The argument names must match the names of the corresponding EOAttribute objects.

For Delete operations, the stored procedure must take an “in” argument for each of the entity’s primary key attributes. The argument names must match the names of the primary key attributes as in a Fetch w/ PK operation stored procedure.

For Get PK operations, the stored procedure must take an “out” argument for each of the entity’s primary key attributes. The argument names must match the names of the primary key attributes as in a Fetch w/ PK operation stored procedure.

Insert, Delete, and Get PK operations should not return a result set.

Modeling Inheritance

One of the issues that may arise in designing your enterprise objects—whether you’re creating a schema from scratch or working with an existing database schema—is the modeling of inheritance relationships.

In object-oriented programming, it’s natural to think of data in terms of inheritance. A Customer object, for example, naturally inherits certain characteristics from a Person object, such as name, address, and phone number. In inheritance hierarchies, the parent object or superclass is usually rather generic so that less generic subclasses of a related type can easily be added. So, in addition to the Customer object, a Client object also naturally derives from a Person object.

While this kind of thinking is inherent in object-oriented design, relational databases have no explicit support for inheritance. However, using Enterprise Objects, you can build data models that reflect object hierarchies. That is, you can design database tables to support inheritance by also designing enterprise objects that map to multiple tables or particular views of a database table.

This chapter discusses when to use inheritance, the different kinds of inheritance supported by Enterprise Objects, and how to implement inheritance. It is divided into the following sections:

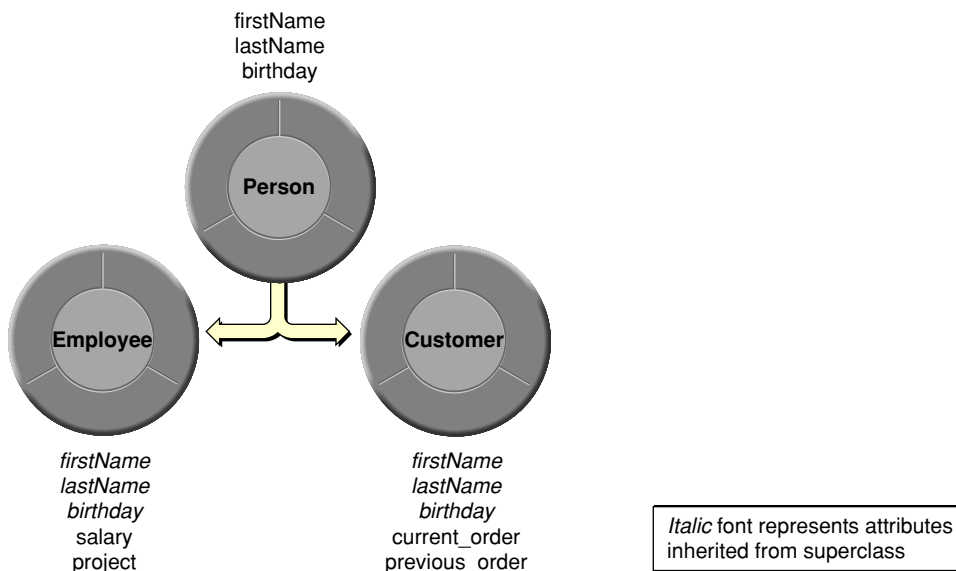
- [“Deciding to Use Inheritance”](#) (page 67) discusses the issues involved when deciding to use inheritance.
- [“Vertical Mapping”](#) (page 69), [“Horizontal Mapping”](#) (page 74), and [“Single-Table Mapping”](#) (page 76) discuss the three approaches to inheritance that are supported by the Enterprise Object technology.

Deciding to Use Inheritance

Using inheritance adds another level of complexity to your data model, data source, and thus to your application. While it has its advantages, you should use it only if you really need to. This section provides information that will help you make that decision.

Suppose you’re designing an application that includes Employee and Customer objects. Employees and customers share certain characteristics such as name and address, but they also have specialized characteristics. For example, an employee has a salary and a manager whereas a customer has account information and a sales contact.

Based on these data requirements, you might design a class hierarchy that has a Person superclass and Employee and Customer subclasses. As subclasses of Person, Employee and Customer inherit Person’s attributes (name and address), but they also implement attributes and behaviors that are specific to their classes, as illustrated in Figure 7-1.

Figure 6-1 A simple object hierarchy

In addition to designing a class hierarchy, you need to decide how to structure your data source so that when objects of the classes are instantiated, the appropriate data is retrieved. These are some of the issues you need to weigh in deciding on an approach:

- Are fetches usually directed at the leaves or the root of the class hierarchy?

When a class hierarchy is mapped onto a relational database, data is accessed in two different ways: by fetching just the leaves (for example, `Employee` or `Customer`) or by fetching at the root (`Person`) to get instances of all levels of the class hierarchy (which includes `Employees` and `Customers`).

- How deep is the class hierarchy?

While deep class hierarchies can be a useful technique in object-oriented programming, you should try to avoid them for enterprise objects. When you attempt to map a deep class hierarchy onto a relational database, the result is likely to be poor performance and a database that's difficult to maintain.

- What is the database storage cost for `null` attributes?

Some approaches to inheritance result in many `null` rows of data.

- Will I need to modify my schema frequently?

Depending on the inheritance approach you take, changes to your database schema can result in maintenance headaches for your enterprise object classes.

- Will other tools be accessing the database?

If other tools write to the data source to which an inheritance hierarchy maps, those tools may not write data in a way that supports the inheritance hierarchy. If this is the case, you should avoid using inheritance to prevent conflicts that may compromise the integrity of your data.

- Can I ensure unique primary keys across inheritance hierarchies?

Within a given inheritance hierarchy, all the primary keys in all the tables must be unique. For example, a primary key value of 36 can occur *only* in one table in an inheritance hierarchy. This may be an issue if you want to apply an inheritance hierarchy to a collection of preexisting database tables that do not have unique primary keys between them.

- Do I need to use inheritance at all?

Don't use inheritance if you're not sure you need it. While a compelling feature, inheritance adds complexity to your application and these costs may outweigh the benefits of using it. In short, use inheritance only if you *need* to—don't use it just because you *want* to.

In object-oriented programming, when a subclass inherits from a superclass, the instantiation of the subclass implies that all the superclass' data is available for use by the subclass. When you instantiate objects of a subclass from database data, all the database tables that contain the data held in each class (whether subclass or superclass) must be accessed so that the data can be retrieved and put in the appropriate enterprise objects.

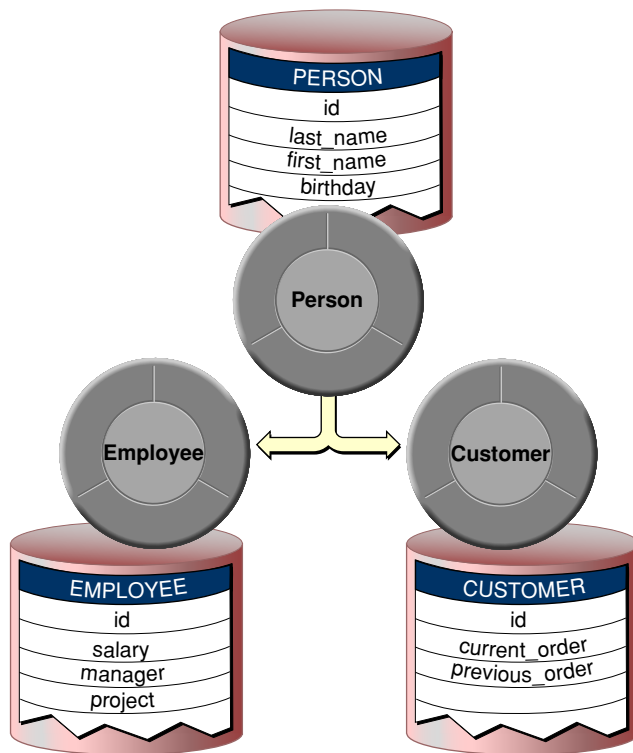
There are different approaches to storing the data in databases for entities that are a part of an inheritance hierarchy. The three approaches supported by Enterprise Objects are

- vertical mapping
- horizontal mapping
- single-table mapping

These approaches, along with the advantages and disadvantages of each, are discussed in the following sections. None of them represents a perfect solution—which one is appropriate depends on the needs of your application. Also keep in mind that you can mix inheritance strategies within a model.

Vertical Mapping

In this approach, each class is associated with a separate table. There is a Person table, an Employee table, and a Customer table. Each table contains only the attributes defined by that class.

Figure 6-2 Vertical mapping

This method of storage directly reflects the class hierarchy. If an object of the Employee class is retrieved, data for the Employee's Person attributes must be fetched along with Employee data. The relationship between Employee and Person is resolved through a join to give Employee access to its Person data. This is also the case for Customer. Vertical mapping requires a restricting qualifier if you want to fetch records from a parent entity (Person in this example).

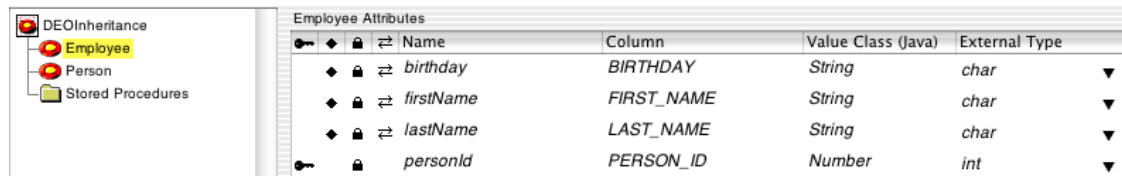
Implementing Vertical Mapping in a Model

Assuming that the entities for each of the participating tables do not yet exist, follow these steps to easily create subtentities from a parent entity:

1. Select the entity you want to be the parent entity (the superclass).

- To create a subentity of the parent entity, choose Create Subclass from the Property menu while the parent entity is selected. Provide a class name and table name for the subentity. In this example, two subentities are added to the model, Employee and Customer, which correspond to two tables you'll create in the database called EMPLOYEE and CUSTOMER, respectively. The attributes a subentity inherits from its parent are displayed in italics, as shown in Figure 6-3.

Figure 6-3 Inherited attributes appear in italics



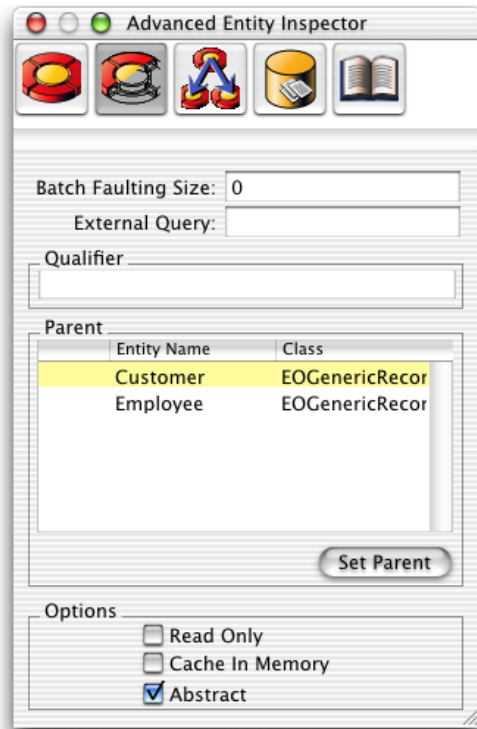
	Name	Column	Value Class (Java)	External Type
	<i>birthday</i>	BIRTHDAY	String	char
	<i>firstName</i>	FIRST_NAME	String	char
	<i>lastName</i>	LAST_NAME	String	char
	personId	PERSON_ID	Number	int

- In the Advanced Entity Inspector, mark the parent entity as abstract if you won't ever instantiate Person objects, as shown in Figure 6-4.

If you need to instantiate the parent entity (Person objects), however, don't mark the parent entity as abstract. If you want to instantiate objects of the parent entity, you also need to assign a restricting qualifier to it. You need to assign a restricting qualifier to any entity in a vertical inheritance hierarchy that is not abstract and that has subentities (leaf nodes).

This is necessary so you can fetch objects of the parent type without also fetching the characteristics of the parent's subentities. That is, when fetching Person objects, you don't also want to fetch attributes in Person's subclasses, Employee and Customer. You do this by assigning a restricting qualifier to the Person entity. See ["Implementing a Restricting Qualifier"](#) (page 78) to learn how to do this.

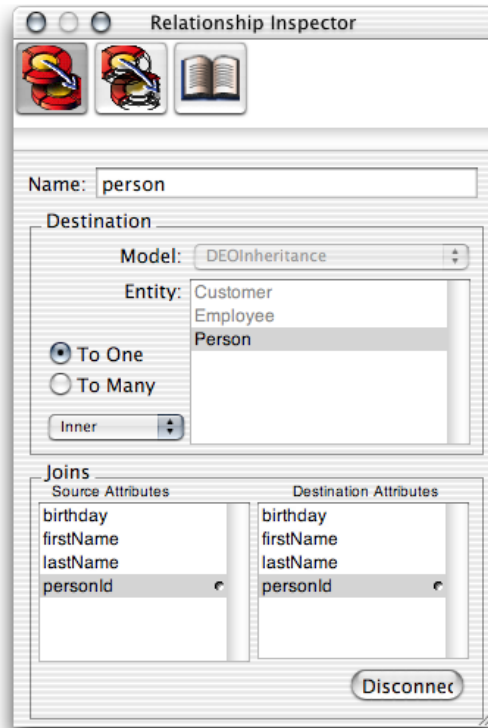
Figure 6-4 Mark parent entities as abstract if they won't ever be instantiated



Assuming that the entities for each of the participating tables already exist, do the following to implement vertical mapping in an EOModel:

1. Create a to-one relationship from each of the child entities (Employee and Customer) to the parent entity (Person) joining on the primary keys. Set the relationships so they are not class properties. Refer to Figure 6-5 for clarity.

Figure 6-5 To-one relationships to parent entity shown in inspector



2. Flatten the Person parent attributes into each child entity (Employee and Customer) and set the flattened attributes as class properties if they are class properties in the Person entity. Do not flatten the primary key. See [“Flattening an Attribute”](#) (page 45) to learn how to flatten an attribute.

If you created the child entities by choosing Create Subclass from the Property menu, you now need to delete the attributes that are inherited from the parent entity. This is necessary to avoid redundancy since the attributes you just flattened reflect the same attributes as the inherited attributes do.

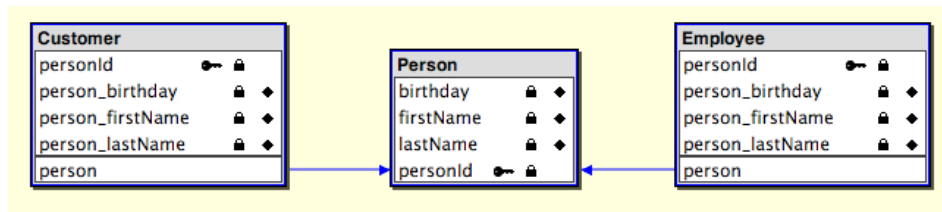
Figure 6-6 shows the result of flattening Person’s attributes into the Employee entity. The flattened attributes appear in bold typeface in the table view.

Figure 6-6 Flattened attributes in table view

Customer Attributes						
	Name	Column	Value Class (Java)	External Type	Value Type	
	<i>personId</i>	<i>PERSON_ID</i>	Number	int	▲	I
◆	person_birthday		String	char	▲	c
◆	person_firstName		String	char	▲	
◆	person_lastName		String	char	▲	

3. Flatten the Person parent entity's relationships into each child entity (Employee and Customer) if it has any relationships, and set them as class properties if they are class properties in the Person entity. In this example, the Person entity has no relationships, so there are none to flatten into its child entities. In diagram view, the three entities should appear as in Figure 6-7.

Figure 6-7 Vertical inheritance hierarchy in diagram view



4. Set the parent entity for each child entity (Employee and Customer) to Person in the Advanced Entity Inspector. This step isn't necessary if you created the Employee and Customer entities using the Create Subclass command from the Property menu.
5. Finally, add attributes to each child entity (Employee and Customer) that are specific to those entities (such as `manager` and `customerSince` in this case).
6. Generate SQL for the Employee and Customer entities to create the EMPLOYEE and CUSTOMER tables in the database.

Advantages of Vertical Mapping

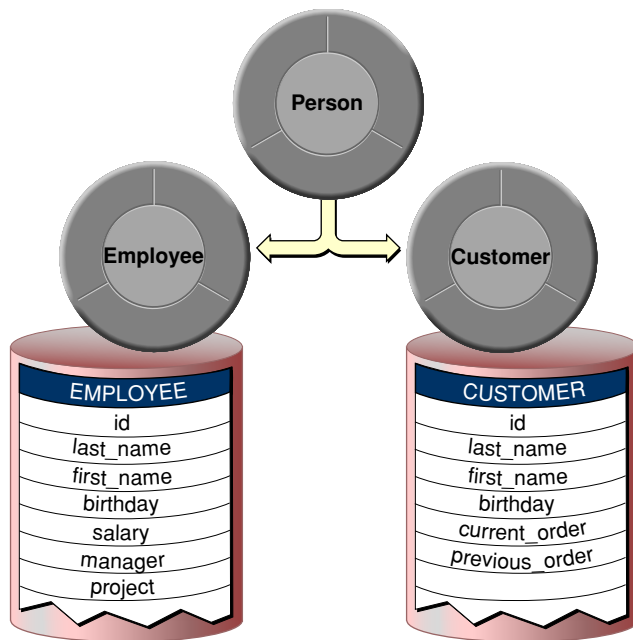
With vertical mapping, a subclass can be added at any time without modifying the Person table. Existing subclasses can also be modified without affecting the other classes in the inheritance hierarchy. The primary virtue of this approach is its clean, “normalized” design.

Disadvantages of Vertical Mapping

Vertical mapping is the least efficient of all the approaches. Every layer of the class hierarchy requires a join table to resolve the relationships. For example, if you want to perform a deep fetch from Person, three fetches are performed: a fetch from Employee (with a join to Person), a fetch from Customer (with a join to Person), and a fetch from Person to retrieve all the Person attributes. If Person is an abstract superclass for which no objects are ever instantiated, the last fetch is not performed.

Horizontal Mapping

In this approach, you have separate tables for Employee and Customer that each contain columns for Person. The Employee and Customer tables contain not only their own attributes, but all of the Person attributes as well. If instances of Person exist that are not classified as Employees or as Customers, a third table would be required. In other words, with horizontal mapping, every concrete class has a self-contained database table that includes all of the attributes necessary to instantiate objects of the class.

Figure 6-8 Horizontal inheritance mapping

This mapping technique entails the same fetching pattern as vertical mapping except that no joins are performed. Horizontal mapping does not require restricting qualifiers.

Implementing Horizontal Mapping in a Model

Assuming that the entities for each of the participating tables do not yet exist, follow these steps to easily create subentities from a parent entity:

1. Select the entity you want to be the parent entity (the superclass).
2. To create a subentity of the parent entity, chose Create Subclass from the Property menu while the parent entity is selected. Provide a class name and table name for the subentity. Refer to [“Implementing Vertical Mapping in a Model”](#) (page 70) for a more concrete example.
3. In the Advanced Entity Inspector, mark the parent entity as abstract if you won’t ever instantiate Person objects, as shown in [Figure 6-4](#) (page 72). Refer to [“Implementing Vertical Mapping in a Model”](#) (page 70) for a more concrete example.
4. Add attributes to each child entity (Employee and Customer) that are specific to those entities (such as `manager` and `customerSince` in this case).
5. Generate SQL for the Employee and Customer entities to create the EMPLOYEE and CUSTOMER tables in the database.

Unlike vertical mapping, you don’t need to flatten any of Person’s attributes into Employee and Customer since they already include all of its attributes.

Advantages of Horizontal Mapping

Similar to vertical mapping, a subclass can be added at any time without modifying other tables. Existing subclasses can also be modified without affecting the other classes in the class hierarchy.

This approach works well for deep class hierarchies as long as the fetch occurs against the leaves of the class hierarchy (Employee and Customer) rather than against the root (Person). In the case of a deep fetch, horizontal mapping is more efficient than vertical mapping since no joins are performed. It's the most efficient mapping approach if you fetch instances of only one leaf subclass at a time.

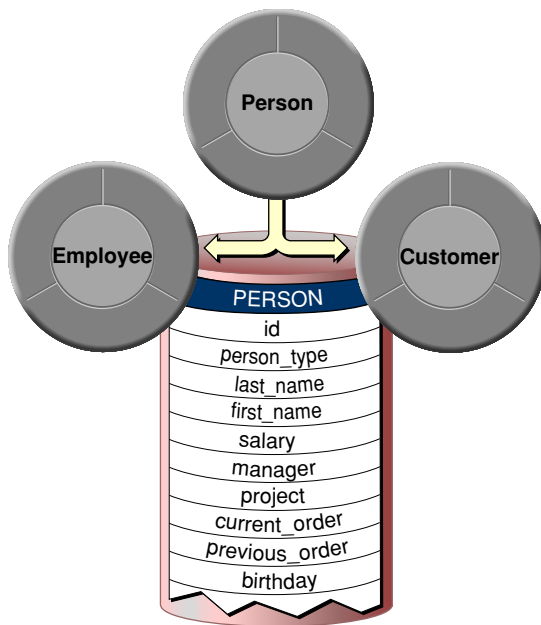
Disadvantages of Horizontal Mapping

Problems may occur when attributes need to be added to the Person superclass. The number of tables that need to be altered is equal to the number of subclasses—the more subclasses you have, the more effort is required to maintain the superclass.

If, for example, you need to add an attribute called `middleName` to the Person class, you then need to alter its subclasses, Employee and Customer. So if you have deep inheritance hierarchies or many subclasses, this can be tedious. However, if table maintenance happens far less often than fetches, this might be a viable approach for your application.

Single-Table Mapping

With single-table mapping, you put all of the data in one table that contains all superclass and subclass attributes. Each row contains all of the columns for the superclass as well as for all of the subclasses. The attributes that don't apply for each object have `null` values. You fetch an Employee or Customer by using a query that returns just objects of the specified type (the table includes a type column to distinguish records of one type from the other).

Figure 6-9 Single-table inheritance mapping

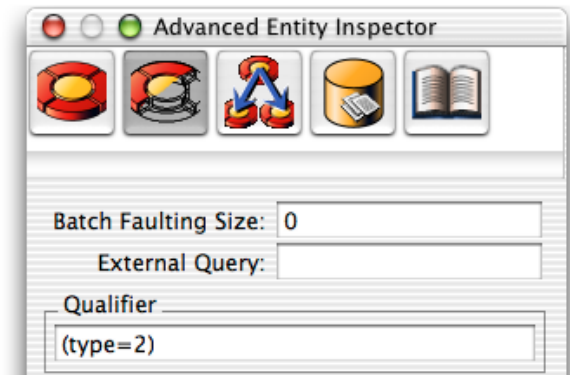
Implementing Single-Table Mapping in a Model

Assuming that the entities for each of the participating tables do not yet exist, follow these steps to easily create subclasses from a parent entity:

1. Select the entity you want to be the parent entity (the superclass).
2. Add an attribute to the parent entity called “type” of External Type `int` and of Internal Data Type Integer. This serves to distinguish each row of data by type. Make this attribute a class property so you can set its value when you insert new objects. Also make sure to add the corresponding column in the database table.
3. To create a subclass of the parent entity, choose Create Subclass from the Property menu while the parent entity is selected. Provide a class name and table name for the subclass. Refer to [“Implementing Vertical Mapping in a Model”](#) (page 70) for a more concrete example.
4. In the Advanced Entity Inspector, mark the parent entity as abstract if you won’t ever instantiate Person objects, as shown in [Figure 6-4](#) (page 72).

5. In the Advanced Entity Inspector, assign a restricting qualifier to the Employee entity that distinguishes its rows from the rows of other entities. Similarly, assign a restricting qualifier to the Customer entity. In this example, you can use `type = 2` for Customer and `type = 9` for Employee. (In [“Implementing a Restricting Qualifier”](#) (page 78) you’ll learn why those two integers are used in this example.

Figure 6-10 Assign a restricting qualifier



Unlike vertical mapping, you don’t need to flatten any of Person’s attributes into Employee and Customer since these entities already have all of Person’s attributes. Each subentity maps to the same table and contains attributes only for the properties that are relevant for that class.

When multiple entities are mapped to a single database table, you must set a restricting qualifier on each entity to distinguish its rows from the rows of other entities. A restricting qualifier maps an entity to a subset of rows in a table. This means that this qualifier is always used when fetches are performed on the entity, as well as any other qualifiers used during the fetch.

The syntax and semantics for restricting qualifiers are the same as for the qualifiers you build in EOModeler. You can use the qualifier builder feature of the fetch specification builder to generate well-formed qualifiers. See [“Building a Qualifier”](#) (page 82).

Implementing a Restricting Qualifier

Finally, for the restricting qualifier to do any good, you need to provide a value for the `type` attribute you added in step 2 for each object you insert (that is, for each record you add to the table). The restricting qualifier uses the `type` attribute, so this example assumes that. (If you use an attribute with a different name to identify rows of data in the table, make the necessary substitutions in this example.)

To provide a value for the `type` attribute for every new object that is inserted in an inheritance hierarchy, you need to:

- define constants for each type
- override `awakeFromInsertion` in a parent enterprise object class
- set the `type` in `awakeFromInsertion`
- return a type in each enterprise object subclass

Consider the `Person` enterprise object parent class. It is an abstract class that has two concrete subclasses, `Employee` and `Customer`. You need to provide constants in the `Person` class to identify these two subclasses:

```
public static final Integer CustomerUserType = new Integer(2);
public static final Integer EmployeeUserType = new Integer(9);
```

Then, you need to override `awakeFromInsertion` in the `Person` subclasses to set the type for each inserted record. An example appears in Listing 6-1.

Listing 6-1 Set type in `awakeFromInsertion`

```
public void awakeFromInsertion (EOEditingContext editingContext) {
    super.awakeFromInsertion(context);
    setType(_userType());
}
```

A subclass (such as `Employee` or `Customer` in this example) must implement the `_userType` method to return an `Integer` representing the object's type:

```
Integer _userType() {
    return CustomerUserType;
}
```

Listing 6-1 assumes that the entity corresponding to the enterprise object class in which the method `awakeFromInsertion` exists includes an attribute named "type" that is a class property. So whenever a new enterprise object is created, its `type` attribute is automatically set to the name of the class.

So if the name of the class is `Customer`, the `type` attribute is set to the integer 2 as soon as the object is created. Then, when a fetch is performed on the `Customer` entity (which is performed on the `Person` table since only one table exists in the database for the objects in this inheritance hierarchy), the restricting qualifier helps to return only those records whose type is "Customer."

See the `WebObjects Examples (/Developer/Examples/JavaWebObjects/)` for a real implementation of this and the other types of inheritance.

Advantages of Single-Table Mapping

This approach is faster than the other two methods for deep fetches. Unlike vertical or horizontal mapping, you can retrieve superclass objects with a single fetch, without performing joins. Adding a subclass or modifying the superclass requires changes to just one table.

Disadvantages of Single-Table Mapping

Single-table mapping results in tables that have columns for all of the attributes of each entity in the inheritance hierarchy. It also results in many `null` row values. While these aren't really disadvantages, they may conflict with some database design philosophies.

Working With Fetch Specifications

After you create and configure entities in EOModeler, you can also use it to build queries on those entities called **fetch specifications**. A fetch specification (a `com.webobjects.eocontrol.EOFetchSpecification` object) is the object Enterprise Objects uses to get data from a data source.

Each fetch specification can contain a qualifier, which fetches only those rows that meet the criteria in the qualifier. A fetch specification allows you to specify a sort ordering to sort the rows of data returned. A fetch specification can also have other characteristics, as discussed in this chapter.

This chapter is organized in the following sections:

- [“Creating a Fetch Specification”](#) (page 81) describes how to use EOModeler to add a fetch specification to an entity.
- [“Assigning a Sort Ordering”](#) (page 85) describes how to assign a sort ordering to a fetch specification.
- [“Prefetching”](#) (page 86) describes what prefetching is and how to configure it.
- [“Configuring Raw Row Fetching”](#) (page 86) describes what raw row fetching is, how to use it, and when to use it.
- [“Other Fetch Specification Options”](#) (page 87) describes the other characteristics of fetch specifications that you can set in EOModeler, such as fetch limit and deep fetching.
- [“Using Named Fetch Specifications”](#) (page 88) tells you how to invoke a fetch specification you build in EOModeler from business logic classes.

Creating a Fetch Specification

Although you can create fetch specifications programmatically, it’s easier and less error-prone to create and configure them in EOModeler. To create a fetch specification in EOModeler:

1. Select the entity with which the fetch specification is associated.

A fetch specification is related to a single entity, so all the fields used in a particular fetch specification are relative to the entity to which it belongs.

2. Choose Add Fetch Specification from the Property menu.

3. Type a name for the fetch specification in the Fetch Specification Name field and press Return.

Figure 7-1 A sample fetch specification

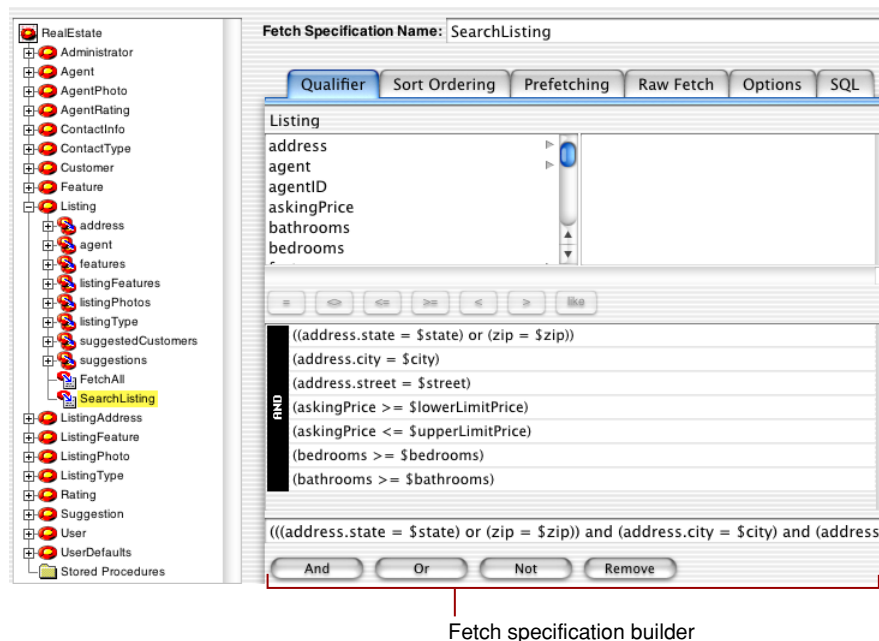


Figure 7-1 shows a fetch specification called SearchListing in an entity called Listing. There are six panes in the fetch specification builder, which you can use to configure a fetch specification. They are each described in the following sections.

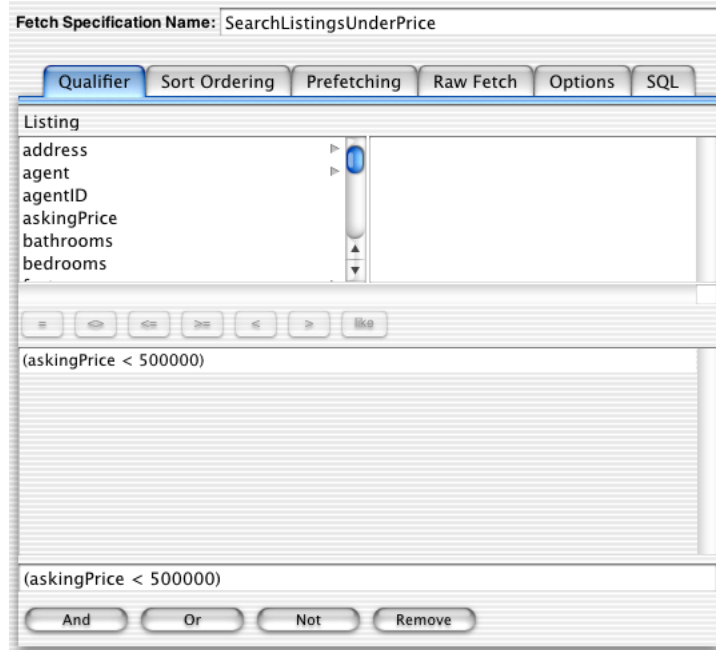
Building a Qualifier

A qualifier (a concrete instance of a `com.webobjects.eocontrol.EOQualifier` subclass) restricts the rows of data that are fetched with a fetch specification object. For example, in a real estate application, you may want to retrieve the records of homes that are priced under \$500,000. To build the qualifier for this query in EOModeler:

1. Using the Real Estate model in the WebObjects examples folder, add a fetch specification to the Listing entity.

2. In the Qualifier pane, click in the text field below the list of attributes, then select the `askingPrice` attribute, and type `< 500000` after it, as shown in Figure 7-2.

Figure 7-2 Static qualifier

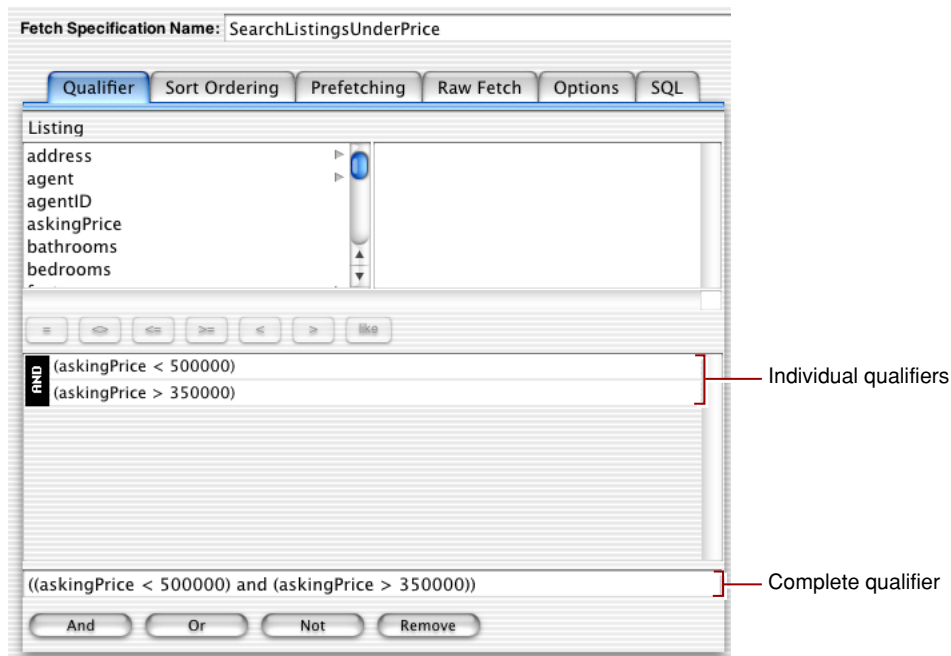


This fetch specification returns only those listings whose asking price is under \$500,000. Although you may have the need to build a static qualifier like this one, you'll most often want to use qualifier variables to supply dynamic values to qualifiers, such as \$350,000 or \$700,000 in this example. This is described in ["Using Qualifier Variables"](#) (page 84).

Creating Compound Qualifiers

You can also use the qualifier builder to create compound qualifiers made up of multiple expressions. For example, you may want to build a qualifier that restricts a query on home listings to homes whose asking price is less than \$500,000 but greater than \$350,000.

To create a compound qualifier from the qualifier created in ["Building a Qualifier"](#) (page 82), select the expression `askingPrice < 500000` and click the **And** button. Then add a second expression by again selecting the `askingPrice` attribute from the attribute list and completing the expression with `> 350000`, as shown in Figure 7-3.

Figure 7-3 A compound qualifier

As you build up a complex query, the text field at the bottom of the Qualifier pane updates to include the full text of the compound qualifier. Instead of building up expressions one by one with the And, Or, and Not buttons, you can type directly into this text field. The qualifier builder parses this string and displays the individual expressions.

Using Qualifier Variables

You can specify static criteria for a fetch specification's qualifier, as is done in ["Building a Qualifier"](#) (page 82) and in ["Creating Compound Qualifiers"](#) (page 83). However, such a qualifier is of limited use. More commonly, you want to specify the form of a qualifier and let users supply specific values when they run the application. You can do this with qualifier variables.

You specify a qualifier variable using the dollar sign character (\$), as in the following:

```
askingPrice < $maxPrice
```

You can build this qualifier in EOModeler as specified in the previous sections and then bind its qualifier variables to your application's user interface. You do this by dragging a fetch specification into either WebObjects Builder or Interface Builder and then binding user interface elements to keys in the display group that correspond to the fetch specification's variables.

Figure 7-4 shows the `queryBindings` attribute of a display group that was added to a `WOComponent`. The `queryBindings` attribute includes bindings for the attributes that are the keys of qualifiers in the fetch specification to which the display group is associated. You bind values in the user interface to these keys by making a connection between the keys and certain dynamic elements.

Figure 7-4 Fetch specification bindings in WebObjects Builder

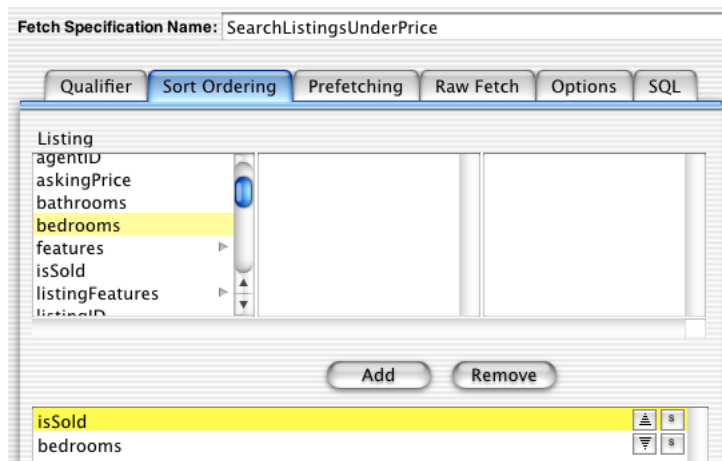
ImageReport		WODisplayGroup		Query Bindings
application		allObjects	>>	bathrooms
session	>	allQualifierOperators	>>	bedrooms
agent	>	batchCount		city
agents	>>	currentBatchIndex		lowerLimitPrice
listing	>	displayedObjects	>>	state
✓ listingDisplayGroup	>	hasMultipleBatches		street
listingPhoto	>	queryBindings	>	upperLimitPrice
listingPhotos	>>	queryMatch	>	zip
listings	>>	queryMax	>	

You can also bind qualifier variables to values in your application programmatically, as described in [“Using Named Fetch Specifications”](#) (page 88).

Assigning a Sort Ordering

It’s common to want a fetch specification to return a sorted array of objects. You can assign a sort ordering to a fetch specification in the Sort Ordering pane of the fetch specification builder.

Simply choose an attribute on which to sort and click Add. The order in which you add attributes specifies the weight to assign to them. Figure 7-5 shows a sort ordering that sorts first on whether the listing is sold and second on the number of bedrooms.

Figure 7-5 Sort ordering

You can also specify an ascending or descending order to sort on for each attribute and whether to perform case-sensitive or case-insensitive comparison. Figure 7-5 specifies an ascending order for the `isSold` attribute and a descending order for the `bedrooms` attribute.

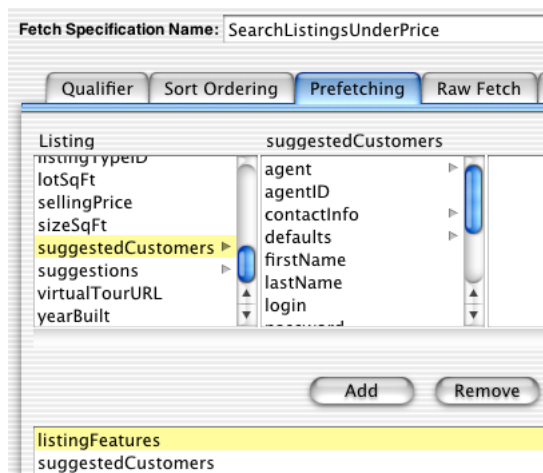
Prefetching

Among the numerous options you can use to tune a fetch specification's behavior is **prefetching**. In the Prefetching pane of the fetch specification builder, you identify relationships that should be fetched along with the objects specified by the fetch specification. For example, when fetching Listing objects, you can prefetch associated `listingFeatures` and `suggestedCustomers` relationships. This tells Enterprise Objects to retrieve a Listing's `listingFeatures` and `suggestedCustomers` relationships along with the Listing itself, as opposed to creating faults for the objects in those relationships.

Although prefetching increases the initial fetch cost, it can improve overall performance by reducing the number of round trips made to the data source.

To specify a relationship to prefetch, in the Prefetching pane of the fetch specification builder, select the relationship you want to prefetch and click Add, as shown in Figure 7-6.

Figure 7-6 Configure prefetching



Configuring Raw Row Fetching

When you perform a fetch in an Enterprise Objects application, the information from the database is fetched and stored in a graph of enterprise objects. This object graph provides many advantages, but it can be large and complex.

If you're creating a simple application, you may not need all the benefits of enterprise objects and the object graph. For example, an application that simply displays information from a database without ever performing database updates and without ever traversing relationships might be just as well served by fetching the information into a set of dictionaries rather than a set of enterprise objects.

You can do this in Enterprise Objects by using **raw row fetching**. In raw row fetching, each row from the database is fetched into an `NSDictionary` object.

When you use raw row fetching, you lose some important features:

- The objects in the dictionary are not uniqued.

- The objects in the dictionary aren't tracked by an editing context.
- You can't access to-many relationship information. (However, you can use key paths to access to-one relationships).

And, if you need an enterprise object at some point in your application, you can easily construct one from a raw row dictionary.

You can configure raw row fetching in the Raw Fetch pane of the fetch specification builder. In this pane, you can choose to fetch all or some of the attributes of a particular entity as raw rows.

Other Fetch Specification Options

The Options pane of the fetch specification builder lets you configure other aspects of a fetch specification. These other aspects are usually used for performance tuning. These are the options:

- **Fetch Limit** lets you specify the maximum number of objects to fetch for a particular fetch specification. You enter the maximum number in the Max Rows text field in the Options pane. The default limit is zero, indicating that there is no fetch limit.

Use the "Prompt on limit" option to specify how Enterprise Objects should behave when the fetch limit is reached. If the "Prompt on limit" option is selected, the user is prompted about whether to continue fetching after the maximum has been reached. If the box isn't checked, Enterprise Objects simply stops fetching when it reaches the limit.
- **Perform deep inheritance fetch** specifies whether to fetch deeply or not. This is used with inheritance hierarchies when fetching for an entity with subentities. A deep fetch produces all instances of the root entity and its subentities while a shallow fetch produces only instances of the entity in the fetch specification. See "[Modeling Inheritance](#)" (page 67) for more details on inheritance.
- **Fetch distinct rows** specifies whether to return distinct results or not. Normally if a record or object is selected several times, such as when forming a join, it appears several times in the fetch results. A fetch specification that fetches distinct rows filters out duplicates so that each record or object appears exactly once in the result set.
- **Lock all fetched objects** specifies that a fetch specification locks fetched objects, which means that each row in the data source is locked when it is read. This is one way to implement pessimistic locking in your application.
- **Refresh refetched objects** specifies that existing objects affected by the fetch specification should be overwritten with newly fetched values when they've been updated or changed. With fetch specifications that don't refresh, existing objects aren't updated when their data is refetched (the fetched data is simply discarded).
- **Require all variable bindings** specifies whether a missing value for a qualifier variable is ignored or whether Enterprise Objects requires that each qualifier variable have a value assigned to it. If this option is selected, an exception is thrown during variable substitution if a missing value is present. If this option isn't selected, any qualifier expressions for which there are no variable bindings are pruned from the qualifier. See "[Using Qualifier Variables](#)" (page 84) for related information.

Using Named Fetch Specifications

Fetch specifications you build in EOModeler are referred to as named fetch specifications. To use a named fetch specification requires that you get hold of the model object in which the fetch specification exists. The code in Listing 7-1 retrieves a fetch specification called “MyFetch” in an entity named “Listing” by looking for it in all the models in the application’s default model group.

Listing 7-1 Get a fetch specification programmatically

```
EOModelGroup modelGroup = EOModelGroup.defaultGroup();
EOFetchSpecification fs = modelGroup.fetchSpecificationNamed("MyFetch",
"Listing");
```

If the fetch specification has qualifier variables, you can bind them to a dictionary of values in the application using the code in Listing 7-2.

Listing 7-2 Bind qualifier variables

```
fs = fs.fetchSpecificationWithQualifierBindings(dictionary);
```

In Listing 7-2, the variable `bindings` is an `NSDictionary` object of key-value pairs. So if you have three qualifier variables in the fetch specification called `$askingPrice`, `$bedrooms`, and `$bathrooms`, the dictionary object would look like Table 7-1.

Table 7-1 Bindings dictionary

Key	Value
askingPrice	500000
bedrooms	4
bathrooms	3

If you define a qualifier like this:

```
(askingPrice < $askingPrice) and (bedrooms = $bedrooms) and (bathrooms =
$bathrooms)
```

then the qualifier variables are replaced by 500000 (`askingPrice`), 4 (`bedrooms`), and 3 (`bathrooms`).

You can construct the dictionary in Table 7-1 with this code:

```
NSMutableDictionary dictionary = new NSMutableDictionary();
dictionary.takeValueForKey("500000", "askingPrice");
dictionary.takeValueForKey("4", "bedrooms");
dictionary.takeValueForKey("3", "bathrooms");
```

The complete code listing for this example appears is:

```
EOModelGroup modelGroup = EOModelGroup.defaultGroup();
EOFetchSpecification fs = modelGroup.fetchSpecificationNamed("MyFetch",
"Listing");
fs = fs.fetchSpecificationWithQualifierBindings(dictionary);
```



```
NSMutableDictionary dictionary = new NSMutableDictionary();  
dictionary.takeValueForKey("500000", "askingPrice");  
dictionary.takeValueForKey("4", "bedrooms");  
dictionary.takeValueForKey("3", "bathrooms");
```


Document Revision History

This table describes the changes to *EOModeler User Guide*.

Date	Notes
2006-05-23	Changed title of book from "WebObjects EOModeler User Guide."
2005-08-11	Changed the title from "Using EOModeler."
2002-11-01	First version of this document.

Glossary

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

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

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

attribute In Entity-Relationship modeling, an identifiable characteristic of an entity. For example, `lastName` can be an attribute of an Employee entity. An attribute typically corresponds to a column in a database table. See also [entity](#) (page 94); [relationship](#) (page 95).

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

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

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

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

column In a relational database, the dimension of a table that holds values for a particular attribute. For example, a table that contains employee records might have a column titled "LAST_NAME" that contains the values for each employee's last name. See also [attribute](#) (page 93).

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

data modeling The process of building a data model to describe the mapping between a relational database schema and an object model.

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

deep fetch An option available to fetch specifications that causes database fetches to occur against the root table and any leaf tables. Applicable to inheritance hierarchies.

derived attribute An attribute in a data model that does not directly correspond to a column in a database. Derived attributes are usually calculated from a SQL expression.

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

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

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

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

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

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

enterprise object A Java object that conforms to the key-value coding protocol and whose properties (instance data) can map to stored data. An enterprise object brings together stored data with methods for operating on that data. It allows this data to persist in memory. See also [key-value coding](#) (page 95); [property](#) (page 95).

entity In Entity-Relationship modeling, a distinguishable object about which data is kept. An entity typically corresponds to a table in a relational database; an entity's attributes, in turn, correspond to a table's columns. An entity is used to map a relational database table to a Java class. See also [attribute](#); [table](#).

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

EOModeler A tool used to create and edit models.

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

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

flattened attribute An attribute that is added from one entity to another by traversing a relationship.

foreign key An attribute in an entity that gives it access to rows in another entity. This attribute must be the primary key of the related entity. For example, an Employee entity can contain the foreign key `deptID`, which matches the primary key in the entity Department. You can then use `deptID` as the source attribute in Employee and as the destination attribute in Department to form a relationship between the entities. See also [primary key](#) (page 95); [relationship](#) (page 95).

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

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

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

inverse relationship A relationship that goes in the reverse direction of another relationship. Also known as a back relationship.

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

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

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

JDBC An interface between Java platforms and databases.

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

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

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

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

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

many-to-many relationship A relationship in which each record in the source entity may correspond to more than one record in the destination entity, and each record in the destination may correspond to more than one record in the source. For example, an employee can work on many projects, and a project can be staffed by many employees. In Enterprise Objects, a many-to-many relationship is composed of multiple relationships. See also [relationship](#) (page 95).

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

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

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

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

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

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

prefetching A feature in Enterprise Object that allows you to suppress fault creation for an entity's relationships. Instead of creating faults, the relationship data is fetched when the entity is first fetched. See also [faulting](#) (page 94).

property In Entity-Relationship modeling, an attribute or relationship. See also [attribute](#) (page 93); [relationship](#) (page 95).

prototype attribute An special type of attribute available in EOModeler to provide a template for creating attributes.

raw row fetching An possible option in a fetch specification that retrieves database rows without forming enterprise objects from those rows.

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

referential integrity The rules governing the consistency of relationships.

reflexive relationship A relationship within the same entity; the relationship's source join attribute and destination join attribute are in the same entity.

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

relationship A link between two entities that's based on attributes of the entities. For example, the Department and Employee entities can have a relationship based on the deptID attribute as a foreign key in Employee, and as the primary key in Department (note that although the join attribute deptID is the same for the source and destination entities in this example, it doesn't have to be). This relationship would make it possible to find the employees for a given department. See also [foreign key](#) (page 94); [primary key](#) (page 95); [many-to-many relationship](#) (page 95); [to-many relationship](#) (page 96); [to-one relationship](#) (page 96).

relationship key A key (an attribute) on which a relationship joins.

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

request A message conforming to the Hypertext Transfer Protocol (HTTP) sent from the user's Web browser to a Web server that asks for a resource like a Web page. See also [response](#) (page 96).

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

response A message conforming to the Hypertext Transfer Protocol (HTTP) sent from the Web server to the user's Web browser that contains the resource specified by the corresponding request. The response is typically a Web page. See also [request](#) (page 96).

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

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

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

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

snapshotting Part of the Enterprise Objects optimistic locking mechanism in which snapshots of database rows in memory are compared with the data in the database.

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

target A blueprint for building a product from specified files in your project. It consists of a list of the necessary files and specifications on how to build

them. Some common types of targets build frameworks, libraries, applications, and command-line tools.

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

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

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

transaction A set of actions that is treated as a single operation that either succeeds completely (COMMIT) or fails completely (ROLLBACK).

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

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

WebObjects Builder A tool used to graphically edit WebObjects components.