Chapter 6

Architectural Overview

Enterprise Objects Framework is a set of tools and resources that helps you create applications that work with most popular relational databases—or with your own custom data store. These tools don't help you build a complete database system from the ground up—the tasks of data storage and retrieval are left to a database server supplied by a third party. Rather, Enterprise Objects Framework lets you design database applications that are easy to build and maintain, that can communicate with other applications, and that draw upon the standard interface features common to all OpenStep applications.

Assuming your data store is a relational database, creating an Enterprise Objects Framework application usually involves the following:

- *A database server and an adaptor for that server.* An adaptor is a mechanism that connects your application to a particular server. For each type of server you use, you need a separate adaptor. Enterprise Objects Framework provides adaptors for Oracle, Sybase, and Informix servers.
- *A model.* A model defines the mapping between your enterprise objects and the server's data; models are most often built graphically using the EOModeler application.
- *The EOPalette.* The EOPalette, used by Interface Builder, gives you access to objects you use in building a user interface.
- *The Enterprise Objects Framework frameworks of classes and protocols.* The classes and protocols provided by the Framework let you programmatically manipulate data as it passes between the server, your objects, and the user interface. Although simple applications can be created entirely in Interface Builder, sophisticated applications will require some use of the Enterprise Objects Framework classes in your own code.

Enterprise Objects Framework

The architecture of the Framework is divided into three major layers, the interface layer, the control layer, and the access layer.

The interface layer provides a standard mechanism for displaying data, the control layer manages a graph of enterprise objects, and the access layer creates enterprise objects from a relational database. The interface layer is connected to the control layer by a *data source*, which supplies the enterprise objects created in the access layer to the interface layer. A data source is an object that has the ability to fetch, insert, update, and delete enterprise objects. It is the means by which the interface layer accesses stored data; from the perspective of the interface layer, how data is stored (whether in a relational database or a flat-file system, for example) is of no consequence. The interface layer interacts with all data sources in the same way.

Enterprise Objects Framework architecture includes the following components:

- The *adaptor level* receives raw data from the database and packages it in dictionary objects.
- The *database level* creates enterprise objects from dictionaries and registers them with the control layer.
- *Models* are used in the access layer to define the mapping between enterprise objects and database data.
- An *EOEditingContext* in the control layer manages a graph of objects and coordinates change notification.
- A *data source* provides the EODisplayGroup with enterprise objects.
- An *EODisplayGroup* (in cooperation with EOAssociations) coordinates the values displayed in the user interface with its enterprise objects and receives change notification from the EOEditingContext.
- *User interface* objects display data from enterprise objects.

Figure 51 shows the architecture of Enterprise Objects Framework.

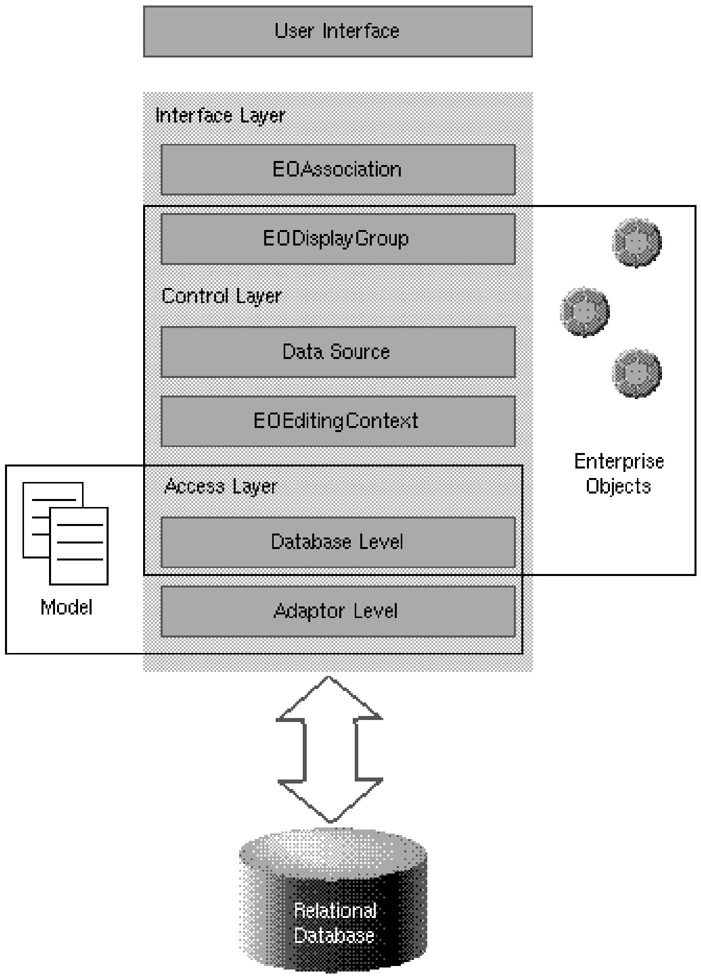

Figure 51. *Enterprise Objects Framework Architecture*

Figure 52 illustrates how data moves through an Enterprise Objects Framework application. Data is packaged differently according to where it is in the Framework. The symbols in the legend indicate the packaging of data at each level of the Framework.

Data flows in an Enterprise Objects Framework application as follows (starting from the bottom of the diagram and working up):

- Data comes into the access layer from a relational database in the form of *rows*.
- The adaptor level packages the raw data as NSDictionary objects. *Dictionaries* contain key-value pairs; each key typically represents the name of a column, and the key's value corresponds to the data for the column in that particular row.
- The database level creates *enterprise objects* from the dictionaries. The enterprise objects' properties get their initial values from the corresponding keys in the dictionary. An enterprise object typically adds behavior to the data it receives from a dictionary.
- The database level registers objects with an EOEditingContext in the control layer.
- The enterprise objects pass from the control layer into the interface layer through a data source, which supplies the objects to an EODisplayGroup.
- The EODisplayGroup notifies EOAssociations that the enterprise objects have new *values*. The EOAssociations take the new values from the enterprise objects and use them to refresh the user interface display.

Movement of data in the Framework is bidirectional: For example, at the user interface level you can fetch data from the database, modify the data, and then update the database to reflect your changes. The repackaging of data at various levels of the Framework is accomplished using reference-counted value classes provided by the Foundation Framework, thereby allowing data to be shared with maximum efficiency.

The primary purpose behind the movement that takes place between the layers of the Framework is to bring together your enterprise objects and persistent data.

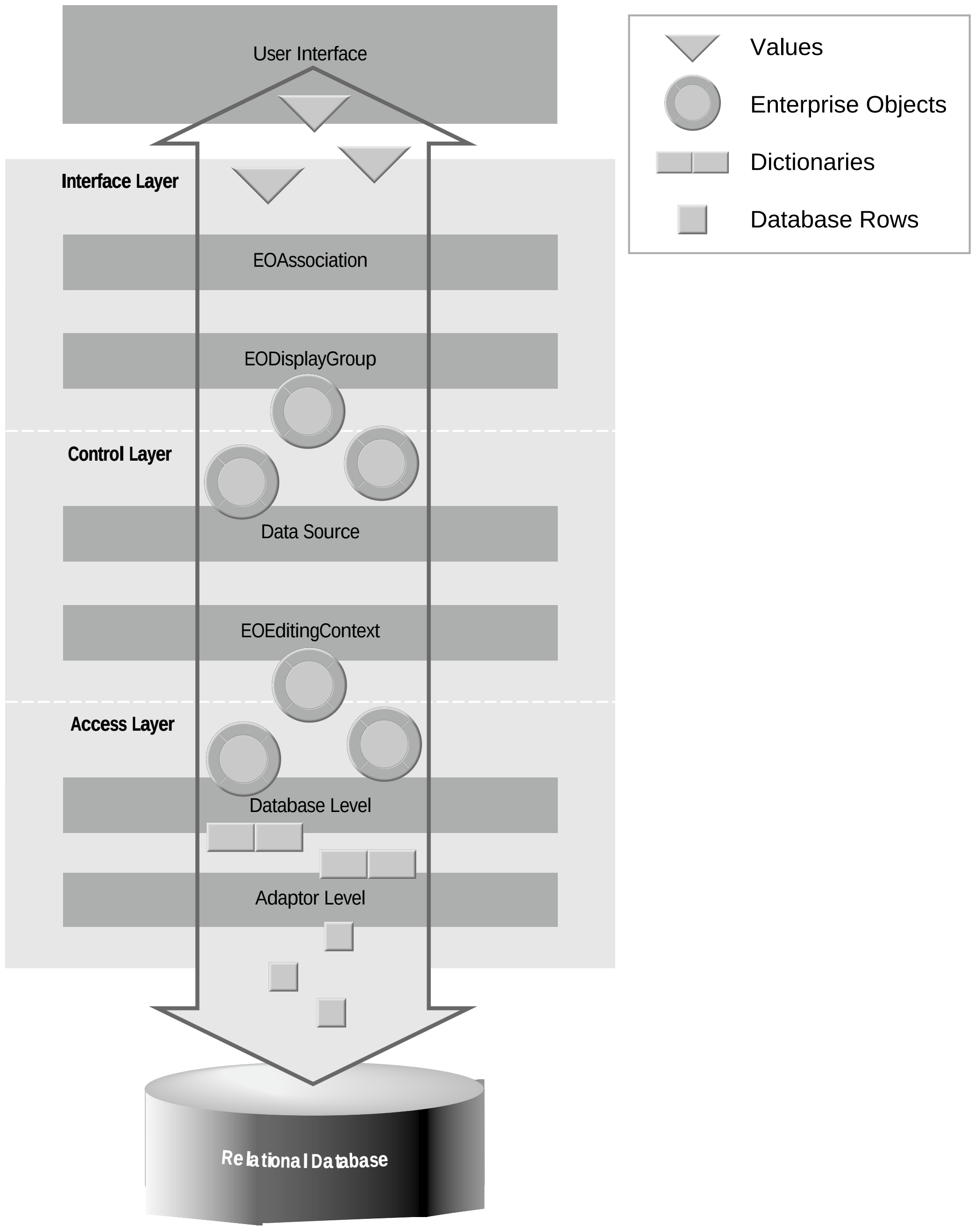

Figure 52. *Data Flow in an Enterprise Objects Framework Application*

Framework Dependencies

The architectural depictions of Enterprise Objects Framework in the previous sections present the ordering of the Framework components in terms of the conceptual data flow in the system. Another way to look at Enterprise Objects Framework is in terms of the structural dependencies of the Framework components on one another.

The control layer is the lowest layer in the Framework. It can be thought of as an extension of Foundation in that it defines generic core functionality, such as key-value access and object change notification. The control layer centers around EOEditingContext, a subclass of EOObjectStore that manages enterprise objects in memory.

The access layer extends the control layer by implementing an EOObjectStore for relational databases, EODatabaseContext.

The interface layer extends the control layer and the Application Kit by adding bindings between enterprise objects and the user interface. This keeps the values of enterprise objects in sync with their display in the user interface.

Figure 53 shows these relationships.

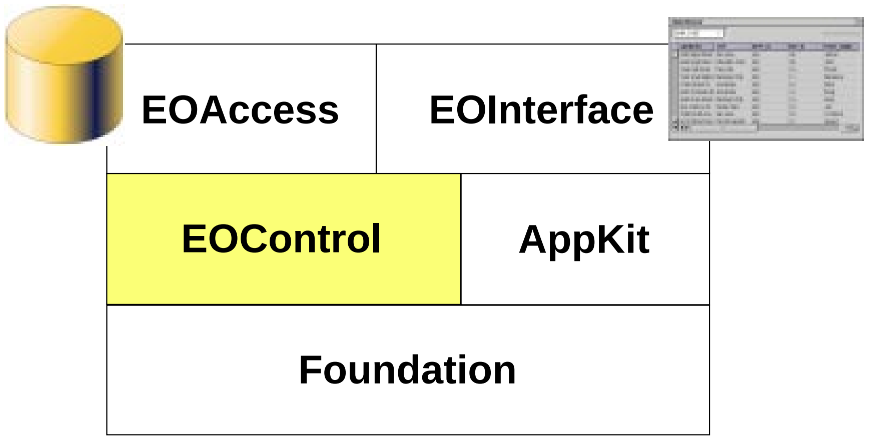

Figure 53. *Framework Dependencies*

What Is an Enterprise Object?

An enterprise object is like any other Objective C object, in that it couples data with the methods for operating on that data. However, an enterprise object class has certain characteristics that distinguish it from other Objective C classes:

- It has properties that map to *stored* data; an enterprise object instance typically corresponds to a single row or record in a database.
- It knows how to interact with other parts of the Framework to give and receive values for its properties.

The ingredients that make up an enterprise object are its class definition and the data values from the database row or record with which the object is instantiated. An enterprise object also has a corresponding model that defines the mapping between the class's object model and the database schema. For more information, see "Models" on page 207.

Enterprise Objects and Data Transportation

EOKeyValueCoding is the means by which data moves through the Framework. Regardless of their other characteristics, objects that conform to the key-value coding protocol (such as enterprise objects) have one thing in common: Their data is accessed by other parts of the Framework as key-value pairs. Key-value coding methods enable an object to receive values for its keys and to give out its keys' values to other parts of the Framework.

By using key-value coding, different types of objects can pass their values to each other, thereby transporting data through the layers of the Framework. When data comes out of the database into the Framework, for example, it's initially packaged in dictionaries from which newly-instantiated enterprise objects get their values (remember, dictionaries are objects that contain data as key-value pairs). Conversely, when data is transported from enterprise objects back to the database, it's repackaged as dictionaries. Note that an enterprise object can itself carry its properties as an NSDictionary object or as regular instance variables; key-value coding applies in either case.

Figure 54 shows how the properties in an enterprise object correspond to the key-value pairs in a dictionary, and how both in turn correspond to a row in a relational database. Enterprise object properties and dictionary keys (such as firstName and lastName) map to columns in the database; the value for each key (for example, "Lesly" and "Oswald", respectively) matches the column's value for the corresponding row.

An enterprise object class doesn't have to map to a single table in a database; it can contain references to multiple tables and have properties for which there are no corresponding database columns. The mapping described in this section refers to the simplest case.

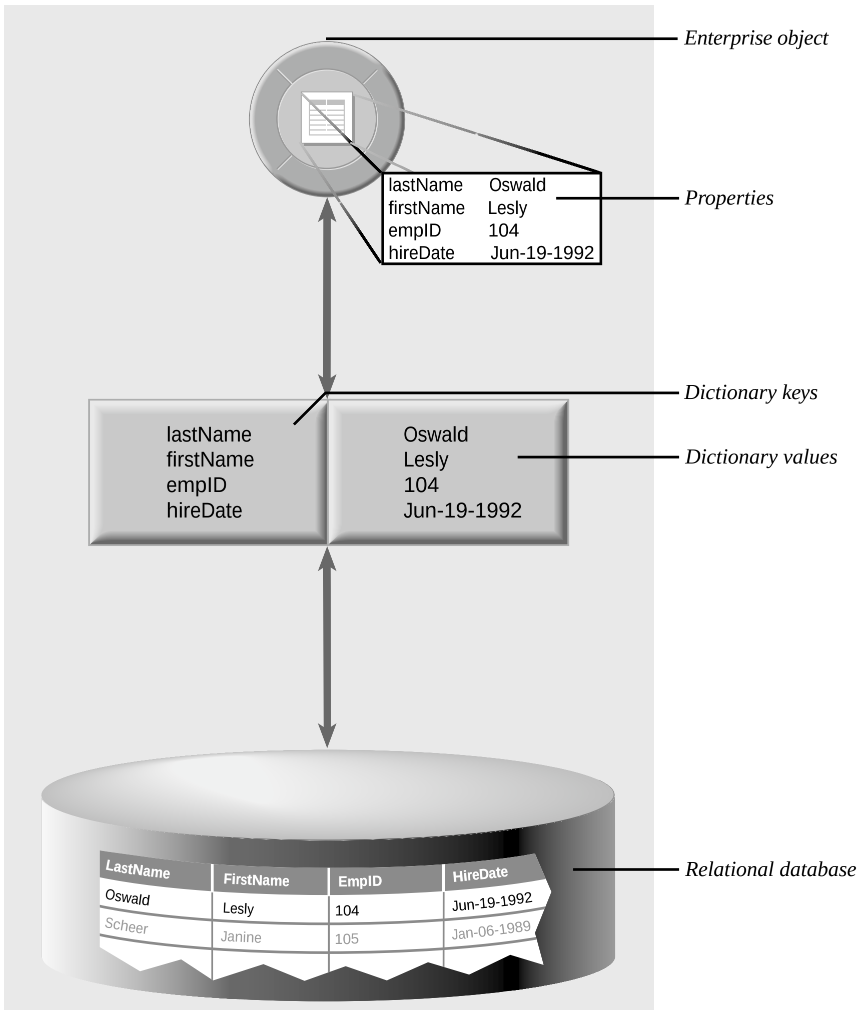

Figure 54. *Enterprise Objects, Dictionaries, and a Relational Database*

An enterprise object can be an instance of either EOGenericRecord or a custom class. EOGenericRecord is the default enterprise object class; a generic record uses an NSDictionary to store its properties, and like custom enterprise objects, conforms to the key-value coding protocol. You use a generic record when you don't need to define special behavior for an enterprise object. A custom class, on the other hand, can carry its properties as a dictionary or as instance variables, but it adds behavior beyond that supplied by key-value coding.

Enterprise Objects Framework Classes

The Enterprise Objects Framework classes are grouped into the following areas:

- User Interface
- Interface Layer
- Data Source (EODatabaseDataSource)
- Control Layer
- Access Layer
- Modeling Classes

Each of these areas and the classes it contains are described in the following sections. Figure 55 shows the classes that play major roles in Enterprise Objects Framework and their relation to each other.

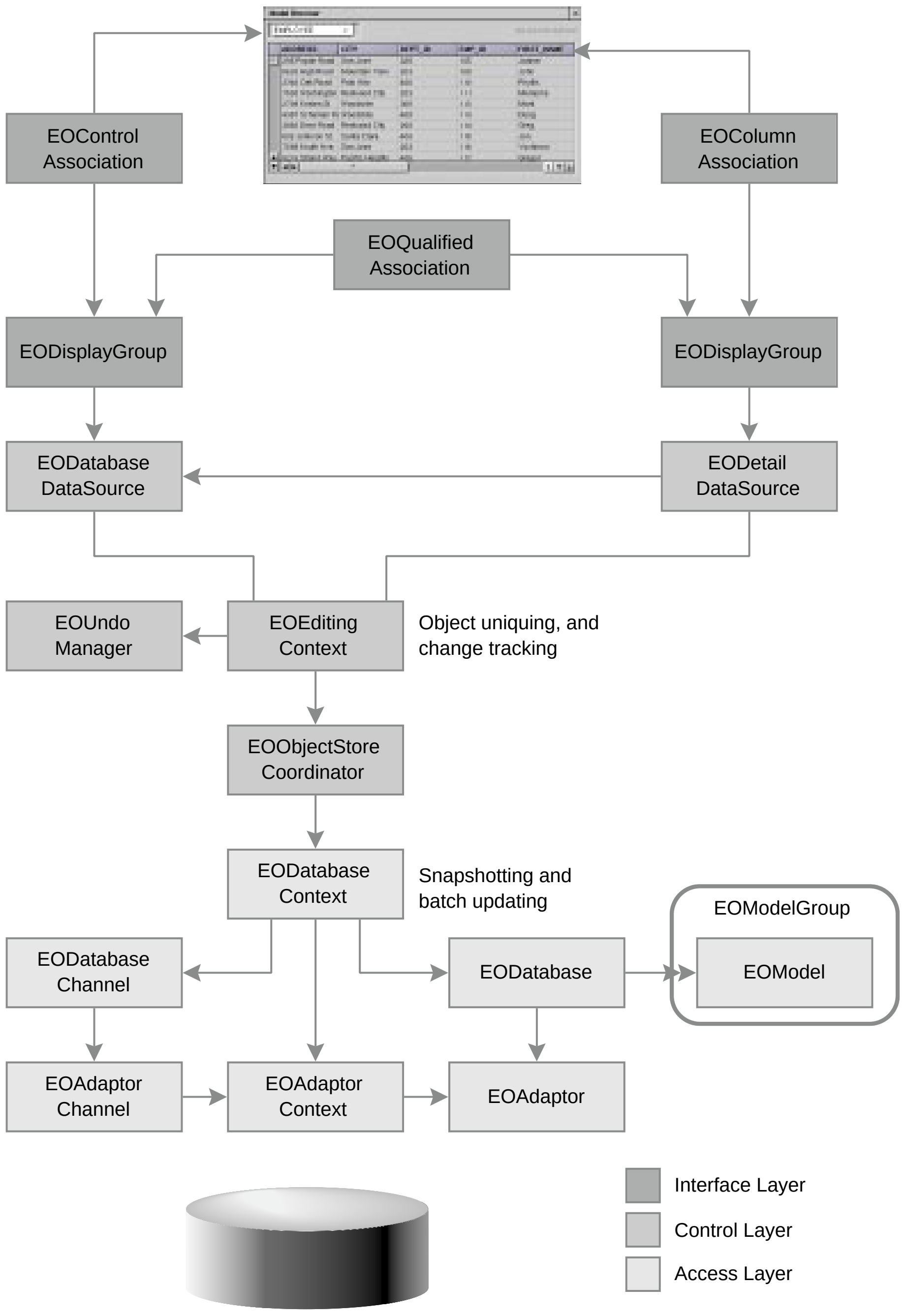

Figure 55. *Enterprise Objects Framework Detailed Architecture*

User Interface Objects

User interface objects such as NSPopUpButtons, NSForms, NSTextFields, and NSTableViews can display the values of enterprise objects, and, if the values are edited in the user interface, communicate the changes back to the enterprise objects.

The Interface Layer

The relationship between user interface objects and enterprise objects is managed by an instance of the EODisplayGroup class. EODisplayGroups are used by EOAssociation objects to mediate between enterprise objects and the user interface. EOAssociations link a single user interface object to one or more class properties (keys) in an enterprise object or objects managed by the EODisplayGroup. The properties' values are displayed in the association's user interface object.

In the Interface layer, EOAssociation objects "observe" EODisplayGroups to make sure that the data displayed in the user interface remains consistent with enterprise object data. EODisplayGroups interact with a data source, which supplies them with enterprise objects.

The Data Source

A data source is an object subclassed from the EODataSource abstract class that presents an EODisplayGroup object with a standard interface to a store of enterprise objects. From the perspective of the EODisplayGroup to which a data source supplies enterprise objects, the actual mechanism used for storing data is of no concern; everything below the data source is effectively a "black box." The interface layer interacts with all data sources in the same way. A data source takes care of communicating with the external data store to fetch, insert, update, and delete objects.

For most database applications, the data source is an instance of the class EODatabaseDataSource or EODetailDataSource (the data source classes supplied with the Framework). EODatabaseDataSource provides an interface to the Framework's access layer and ultimately, to a relational database. However, the data source can be any object subclassed from EODataSource. Thus, the user interface layer can be used independently from the access layer for other types of data sources, such as an array of objects constructed by an application, or objects fetched from a flat-file database or a newsfeed.

The Control Layer

Within an Enterprise Objects Framework application, enterprise objects are the focal point. They encapsulate the most current data for your application (including data that hasn't been committed to the database yet), and the business logic for operating on that data.

The control layer facilitates the central role of enterprise objects by providing an infrastructure for them that is independent of the user interface and the storage mechanism being used. From a development standpoint, this means that you can use the classes in the control layer to write enterprise objects that have no dependencies on the interface layer or the access layer. The control layer dynamically manages the interaction between these objects and the rest of your application by:

- Tracking changes to enterprise objects
- Updating the user interface when object values change
- Updating the database when changes to objects are committed
- Managing undo in the object graph
- Managing uniquing

Uniquing is used in the Framework to uniquely identify enterprise objects and maintain their mapping to stored data. Enterprise objects have a *primary key*, which is defined in the model that maps the object to the database. This primary key is used to maintain the identification between an enterprise object instance and a corresponding database row. Uniquing is also used to ensure that if an object already exists in memory, another instance of it isn't created when a row with the same primary key is fetched from the database. So, for example, if two employee objects have the same manager, a single instance of the manager object resides in memory, and both employee objects refer to it.

The control layer's major areas of responsibility and the classes involved are described in the following table:

Responsibility	Classes
Object Graph Management	EOEditingContext EOUndoManager EOObserver
Object Storage Abstraction	EOObjectStore EOGlobalID EOFault
Object Query Specification	EOQualifier EOSortOrdering EOFetchSpecification
Protocols to interface with enterprise objects	Validation (EOClassDescription) EOKeyValueCoding
Simple Source of Objects (for EODisplayGroup)	EODataSource EODetailDataSource

Because they constitute major conceptual pieces of the Enterprise Objects Framework architecture, object graph management and the object store abstraction are discussed in more detail in the following sections.

Object Graph Management

An object graph is a group of related business objects that represent an internally consistent view of one or more external stores. In a running application, the object graph is the central repository for data and business logic. The class that plays the most significant role in object graph management is EOEditingContext.

EOEditingContext

An EOEditingContext, which represents a single "object space" or document in an application, manages an in-memory graph of enterprise objects. An EOEditingContext can also be thought of as a transaction scope. All objects fetched from an external store are registered in an EOEditingContext along with a global identifier (EOGlobalID) that's used to uniquely identify each object to the external store. The EOEditingContext is responsible for watching for changes in its objects (using the EOObserving protocol) and recording snapshots for object-based undo.

The object graph that an EOEditingContext monitors is created by the EOEditingContext's parent EOObjectStore. The EOEditingContext is itself an EOObjectStore, which gives it the ability to act as an EOObjectStore for another EOEditingContext. In other words, EOEditingContexts can be nested, thereby allowing a user to make edits to an object graph in one editing context and then discard or commit those changes to another object graph (which, in turn, may commit them to an external store). A single enterprise object instance exists in one and only one context, but multiple copies of an object can exist in different EOEditingContexts. Thus object uniquing is scoped to a particular EOEditingContext. For more information, see the EOEditingContext class specification in the *Enterprise Objects Framework Reference*.

Object Storage Abstraction

The class that plays the most significant role in the Enterprise Objects Framework storage abstraction is EOObjectStore.

EOObjectStore

EOObjectStore defines a abstract class for objects that act as an “intelligent” source and sink of objects for an EOEditingContext. The object store is responsible for constructing and registering objects, servicing object faults, and committing changes made in an EOEditingContext.

Some of the subclasses of EOObjectStore are EOEditingContext (for use in nested EOEditingContexts), EODatabaseContext (for mapping objects from a relational database), and EOObjectStoreCoordinator (for coordinating multiple object stores).

The Access Layer

The access layer allows your application to interact with database servers at a high level of abstraction. The access layer is divided into two parts:

- A *database level* that allows applications to treat records as full-fledged enterprise objects.
- An *adaptor level* for server-independent access to records that don’t have custom behavior.

Working with the access layer allows you to have a finer level of control over database operations.

The top row of classes in Figure 56 (EODatabase, EODatabaseContext, and EODatabaseChannel) constitutes the database level. The bottom row of classes (EOAdaptor, EOAdaptorContext, and EOAdaptorChannel) constitutes the adaptor level. EOModel objects are used by the access layer to log into a database server and establish the mapping between enterprise objects and database data. The database level, adaptor level, and models are described in the following sections.

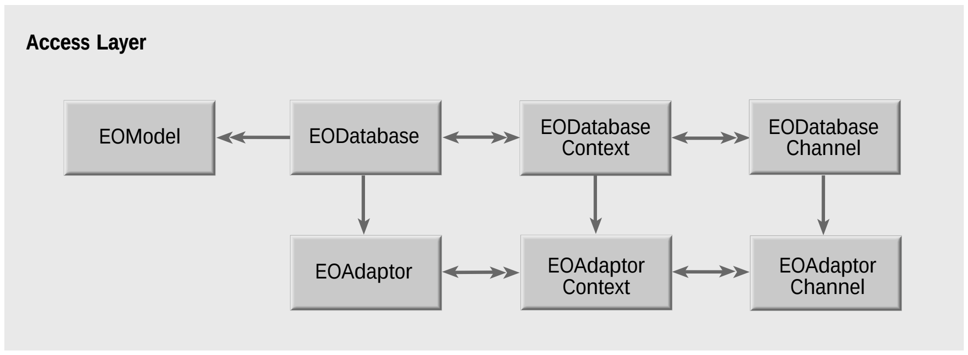

Figure 56. *The Access Layer*

The Database Level

The database level is where enterprise objects are created from the dictionaries retrieved by the adaptor level. It's also where snapshotting is performed. *Snapshotting* is used by Enterprise Objects Framework to manage updates. When an object is fetched from the database, a snapshot is taken of its state. A snapshot is an NSDictionary object; it's consulted when you perform an update to verify that the data in the row to be updated has not changed since you fetched the object.

Figure 57 shows the database level classes and the behaviors associated with each class.

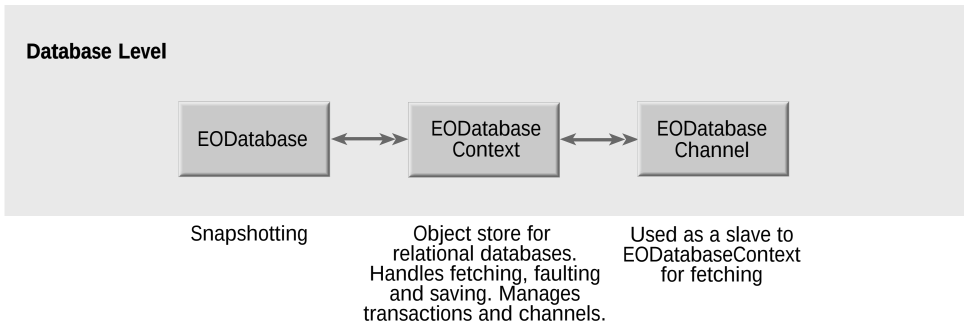
Database Level
EODatabase
EODatabase Context
EODatabase Channel
Snapshotting
Object store for relational databases. Handles fetching, faulting and saving. Manages transactions and channels.
Used as a slave to EODatabaseContext for fetching

Figure 57. *Database Level*

The Adaptor Level

While the database level deals with data packaged as enterprise objects, the adaptor level deals with database rows packaged as dictionaries. An adaptor is the mechanism through which your application communicates with a database server.

Figure 58 shows the database level classes and the behaviors associated with each class.

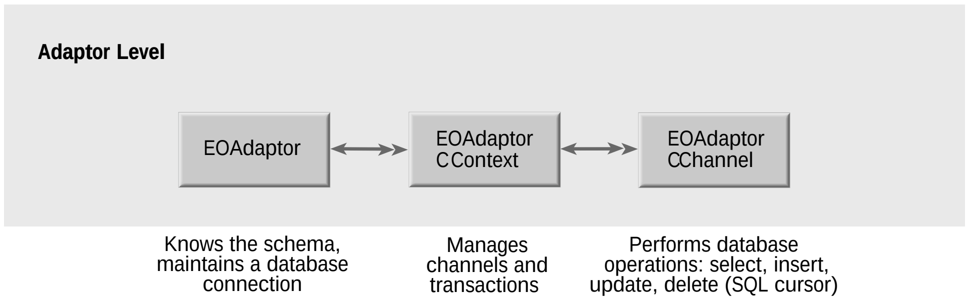
Adaptor Level
EOAdaptor
EOAdaptor CContext
EOAdaptor CChannel
Knows the schema, maintains a database connection
Manages channels and transactions
Performs database operations: select, insert, update, delete (SQL cursor)

Figure 58. *Adaptor Level*

The adaptor level classes define a server-independent interface for working with relational database systems. Server-specific subclasses encapsulate the behavior of database servers, thereby offering a uniform way of interacting with servers while still allowing applications to exploit their unique features.

Models

The correspondence between an enterprise object class and stored data is established and maintained by using a *model*. A model defines, in entity-relationship terms, the mapping between enterprise object classes and a physical database. Figure 59 shows the modeling classes, including the class EOJoin. Join objects identify the entities and attributes that are linked by a relationship.

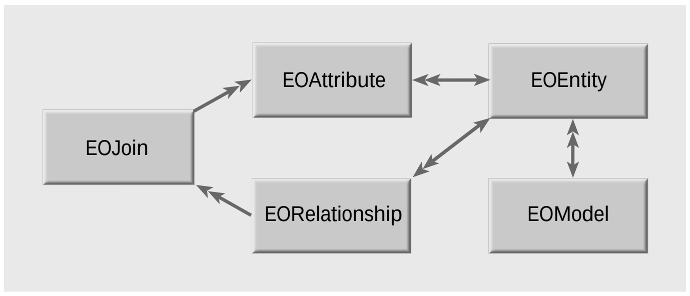

Figure 59. *Modeling Classes*

The following table describes the database-to-object mapping provided in a model:

Database Element	Class
Data Dictionary	EOModel
Table	EOEntity
Column	EOAttribute
Row	Enterprise object class

While a model can be generated at run time, the most common approach is to use the EOModeler application to create models that can be stored as files and added to a project. You use a model throughout the development and deployment of your application to maintain the mapping between enterprise objects and persistent data.

In addition to storing a mapping between the database schema and enterprise objects, a model file stores information needed to connect to the database server. This connection information includes the name of an adaptor bundle to load so that Enterprise Objects Framework can communicate with the database.

For a discussion of entity-relationship modeling and how it relates to Enterprise Objects Framework, see the appendix “Entity-Relationship Modeling.”