# ENTERPRISE OBJECTS™ FRAMEWORK: BUILDING REUSABLE BUSINESS OBJECTS

# The Enterprise Objects™ Framework

*Executive Summary*

**The NeXT Enterprise Objects Framework couples relational databases with objects in the NEXTSTEP application environment. Using the EO Framework, developers can forge enterprise *data* into something far more valuable: enterprise *objects*. Enterprise objects, such as customer and sales order, integrate business data with the business *policies* that makes the data meaningful: a customer credit object defines not only a credit rating, but the rules by which it is computed. Enterprise objects do not change the structure of existing databases or threaten the applications that depend on them. The EO Framework works today with Oracle and Sybase databases, and its design is open to others.**

**Enterprise objects are first class citizens of the most respected object-oriented user and developer environment: NEXTSTEP. They are integrated with the Project Builder and Interface Builder graphical development tools, they can be reused in application after application; they can be refined by subclassing. And they can be distributed among heterogeneous servers with NeXT's Portable Distributed Objects system.**

**As is usual in the NEXTSTEP environment, simple enterprise object applications can be constructed without coding. Developers with special requirements will find an accommodating architecture "under the hood" that separates functions so developers don't have to understand everything to do anything. Even the most sophisticated applications can begin as simple ones, getting the basics working first without programming, then coding additional features in easy-to-manage increments.**

## I. Business Perspective

This paper provides two views of the Enterprise Objects Frameworks for two kinds of readers. This Business Perspective section describes the business advantages of the Enterprise Objects Framework. The Technical Perspective section starting on page 5 describes development with the EO Framework and some of its internal design and operation.

### The Policy-data Split

Databases are excellent data stores. They are fast and robust, they support concurrent and selective access and data distribution. But enterprises don't run on just data. Data gets its meaning from associated business practices and policies. For example, a customer credit rating (data) has little meaning separate from the policy that determines credit ratings. And a credit rating is erroneous if it has been computed according to an obsolete policy.

With so many enterprises almost continuously re-engineering their business practices, policy changes are frequent. But it is hard to keep database applications synchronized with the changes because the data and the software representation of policy are separated. Although the data is centrally stored, the policy is often replicated across multiple database programs and screens created with 4th generation languages (4GLs). When the policy for computing credit changes, how can one be sure that all screens that compute credit ratings are updated? If they aren't, some ratings will be computed the new way, some the old—with no way for users to tell the difference. Policy isn't policy when it's inconsistently applied.

### Objects Fuse Policy and Data

Object-oriented software development has entered the mainstream in the 1990s. One of its major appeals is the fusing of policy (called methods) with data (called instance variables) into packages called objects (see Figure 1). In a customer credit object, for example, a credit rating and the method that computes it are inseparable. Changing the policy changes the data the next time the object is used. (Readers familiar with user interface button objects may recall how a change to a button's implementation takes effect in all applications that use it.)

Integrating policy and data makes software easier to understand and modify because associated policies and data are defined in a single source

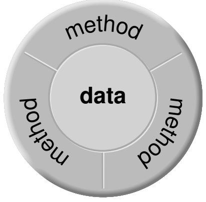



*Figure 1 An object fuses policies and data as methods and instance variables*

file—no matter how many applications use the object. Indeed, reuse is another attraction of objects; an object that embodies policy and data can be used by many applications, ensuring consistency across enterprise functions and divisions. Integrating policy and data can also increase user confidence by ensuring that data is meaningful and trustworthy.

## Another Gap: Relational Databases and Objects

Many new object-oriented applications have been successfully deployed in the past few years. Hundreds of them have been based on NEXTSTEP. But despite its potential for eliminating the policy-data split, object-oriented development has had little impact on enterprise information systems departments. These departments manage relational databases, which are immensely valuable information resources. These resources must be preserved because critical applications depend on them. At the same time, existing data could be employed to much greater effect if it could be coupled with object-oriented development. It is little surprise that considerable attention has, and is, being paid to bridging the gap between objects and relational databases.

Bridging the gap is one thing, bridging it well is another. NeXT's Database Kit™ was an early effort, one which provided the experience necessary to develop its successor, the Enterprise Objects Framework. Other approaches examine a database and generate source code that inserts database records into an object wrapper. This approach leaves developers with another kind of legacy: they have to understand and maintain the generated code forever. An alternative approach—the one used by the EO Framework—is to generate objects dynamically at run-time from developer-supplied definitions of data and methods. Developers can change these definitions at any time; the changes take effect the next time the objects are created from the database. In the EO Framework, the only code to be maintained is the developer-written methods that capture business policies.

Bridging the gap between databases and objects alone will not ensure that quality applications can be rapidly developed and rapidly adapted to changing business conditions. Rapid application development and adaptation requires more than objects. They require a rich environment that supports all facets of application development; an environment such as NEXTSTEP.

## What is the Enterprise Objects Framework?

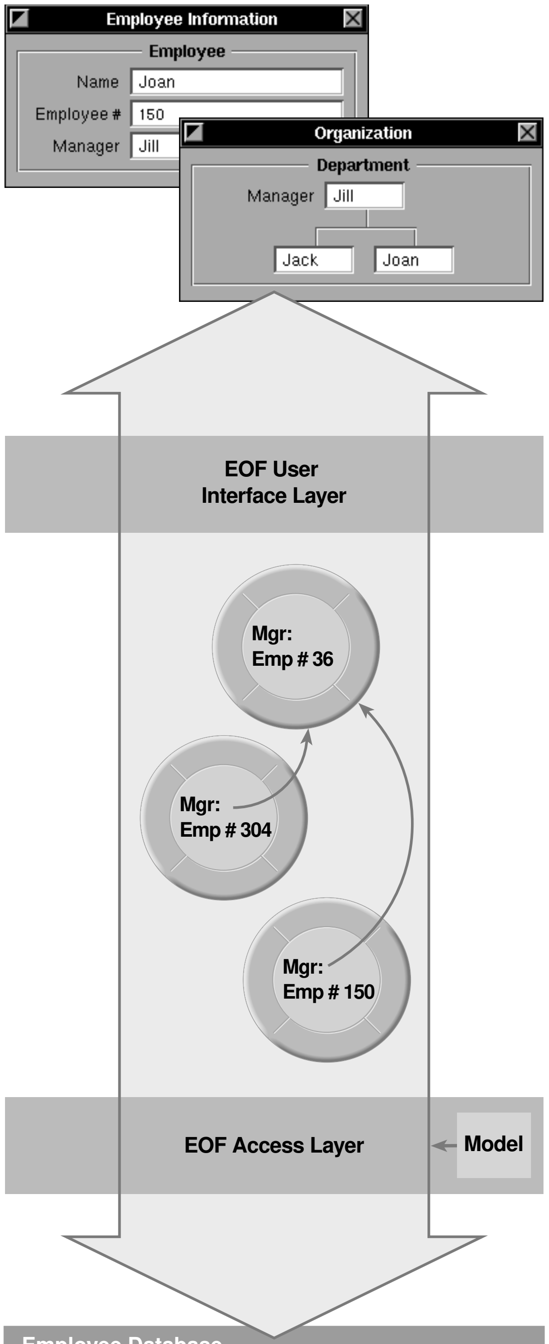
NeXT's Enterprise Objects Framework links the strength of relational database storage with object-oriented development and integrates them in the world's most respected object-oriented user and development environment: NEXTSTEP.

Figure 2 summarizes how the Enterprise Objects Framework transforms rows in a database table into enterprise objects, and saves user changes to the objects back to the database. An EO Framework-based application is like any other NEXTSTEP application; it may display multiple windows, formatted text fields, images, sounds, and so on. In response to user actions, user interface objects (buttons, and so on) send messages to the EO Framework user interface layer. The user interface layer communicates these actions to enterprise objects, and directs the access layer to fetch new objects and to save changed ones. The EO Framework access layer creates enterprise objects in memory from database rows and saves changes to those objects back to the database. The access layer communicates with the database server by means of SQL, and does not alter the structure of the database. To create an enterprise object, the access layer consults a developer-supplied model, which specifies database-to-object data correspondences.

## Enterprise Objects are Reusable

Reuse has become a buzzword, but that doesn't diminish its validity. An excellent way to deploy new applications faster is to do less work by reusing the work of others. Data reuse is a familiar notion in the database world, where the same tables are often used by multiple applications. Code reuse is rare in conventional database applications. Most code is tightly bound to screens, rather than stored in common libraries. Although stored procedures can be reused, they are written in languages that have limited applicability; they are not suited for serious math, for example. Databases are principally storage engines, not computation engines.

Enterprise objects, coupling methods and data, allow both code and data to be reused. The same vendor object, for example, can be used by manufacturing, accounts payable, and tax reporting applications. Moreover, enterprise objects support what might be termed "selective reuse." Suppose a business has developed a class that represents a customer. That class can be fully reused across many applications.




**Employee Database**

| emp. # | name | title | mgr |
|---|---|---|---|
| 36 | Jill | Pres. | |
| 304 | Jack | VP Sales | 36 |
| 150 | Joan | Treas. | 36 |

*Figure 2 Data flows from database tables to enterprise objects to the application user interface and back.*

But suppose as well that the business later decides to treat residential customers differently from business customers in a few ways.

To realize this new policy requires the addition of ResidentialCustomer and BusinessCustomer classes to the existing Customer class. Notice the phrasing: "in addition to," not "in place of"; that's because in most ways residential and business customers are still just customers. With enterprise objects, developers can define ResidentialCustomer and BusinessCustomer as subclasses of Customer. These subclasses automatically inherit the Customer methods (policies); the only new programming they need implements the *differences* between residential and business. The new ResidentialCustomer and BusinessCustomer subclasses reuse most of the tested code of the Customer class. (Notice as well how development of the new subclasses can proceed incrementally; from the outset ResidentialCustomer objects behave like Customer objects; their behavior can be changed in small steps, one method at a time.) Isn't this the way programming is *supposed* to be?

When people think of reuse they normally think "reduced development time," but there are other advantages as well:

- Reused classes are more trustworthy than new ones because they have been proven in other applications.
- Reused classes mean consistent application of policy across the enterprise.

## Enterprise Object-based Applications are Scalable

One drawback to the most common means of database application development—4GLs—is their single source of processor cycles: they must all come from the client machine. The more compute-intensive the application, the less responsive is its user interface. The only way to improve response is to upgrade the client machine.

In contrast, enterprise object-based applications can be distributed across multiple servers, allocating compute-intensive objects to servers built for the purpose, leaving the client machines to mainly manage the user interface. Enterprise objects, like all NEXTSTEP objects, can be distributed to NEXTSTEP-based servers with the Distributed Objects facility. They can also be distributed to other kinds of servers (for example, HP and Sun) that are running NeXT's Portable Distributed Objects

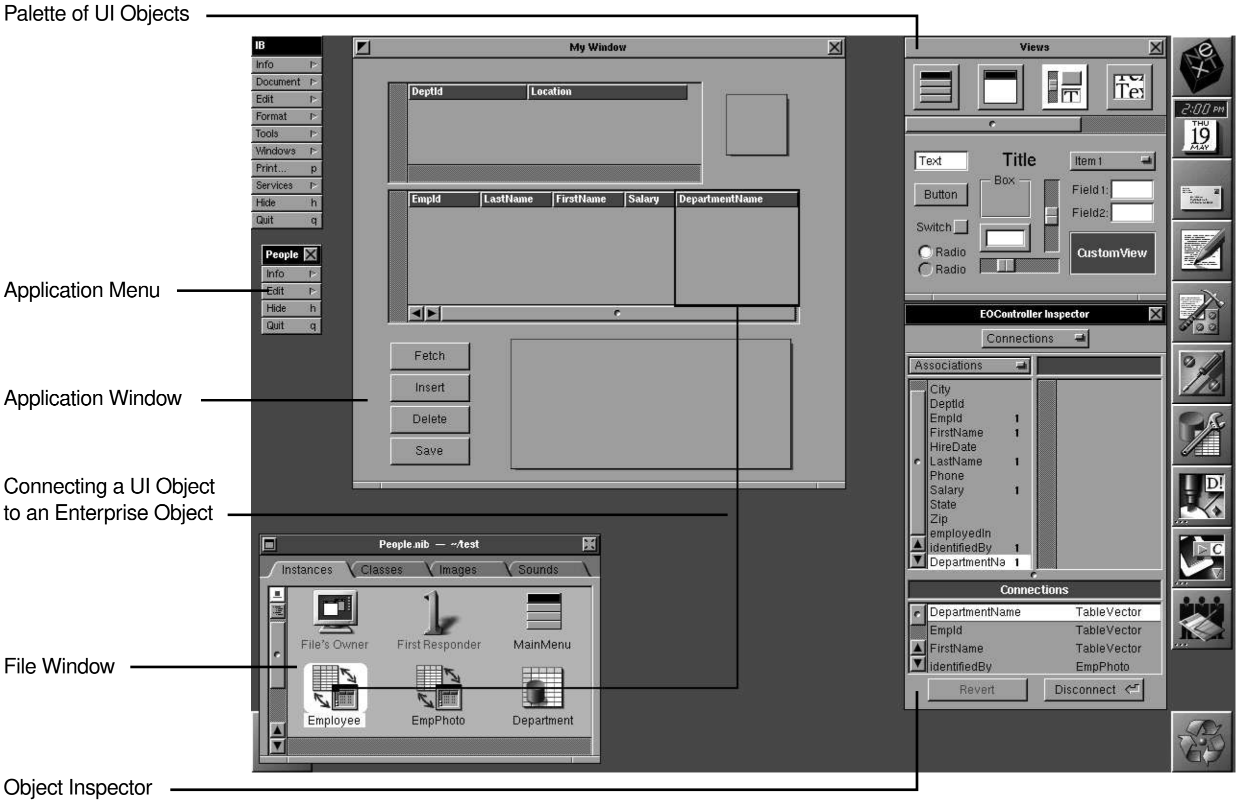



*Figure 3 An EO Framework-based application in the early stages of construction with the NEXTSTEP Interface Builder*

software. It is not only enterprise objects that can be distributed; the objects that constitute the enterprise object framework can also be allocated to the machines where they are executed most effectively.

### Perfect NEXTSTEP Integration

Enterprise objects, and applications that use them, are developed in the same NEXTSTEP environment that has brought five-fold productivity improvements to 100,000 customers. Enterprise objects are full-fledged citizens of the NEXTSTEP run-time environment; they can do anything that other objects can do.

Figure 3 suggests what enterprise object-based application development is like with the NEXT-STEP Interface Builder. The snapshot was taken about 10 minutes into the development of a simple master-detail style employee application. In the completed application, departments will be listed in the upper table of the application window; clicking on a department will list its employees in the table beneath. Clicking on an employee will display his or her photograph in the upper right corner of the application window, and a graphical depiction of the employee's location in the lower part of the window.

All of the user interface objects in the application window were created by dragging from UI palettes, dropping on the window, then positioning and resizing as in an object-oriented drawing program. The developer has just connected the Employee enterprise object icon in the File window to the DepartmentName column of the employee table, indicating by this gesture that Employee enterprise objects are the source of department names.

Although this is a simple application—essentially displaying data from EOs—it will be developed without writing a line of code. It is a good example of the power of object reuse: the Employee EOs and the user interface and infrastructure objects are supplied by the Interface Builder and the EO Framework.

From the moment an Interface Builder-developed application is created, it is executable and can be tested within Interface Builder itself. Developer-supplied code takes the form of additional objects,

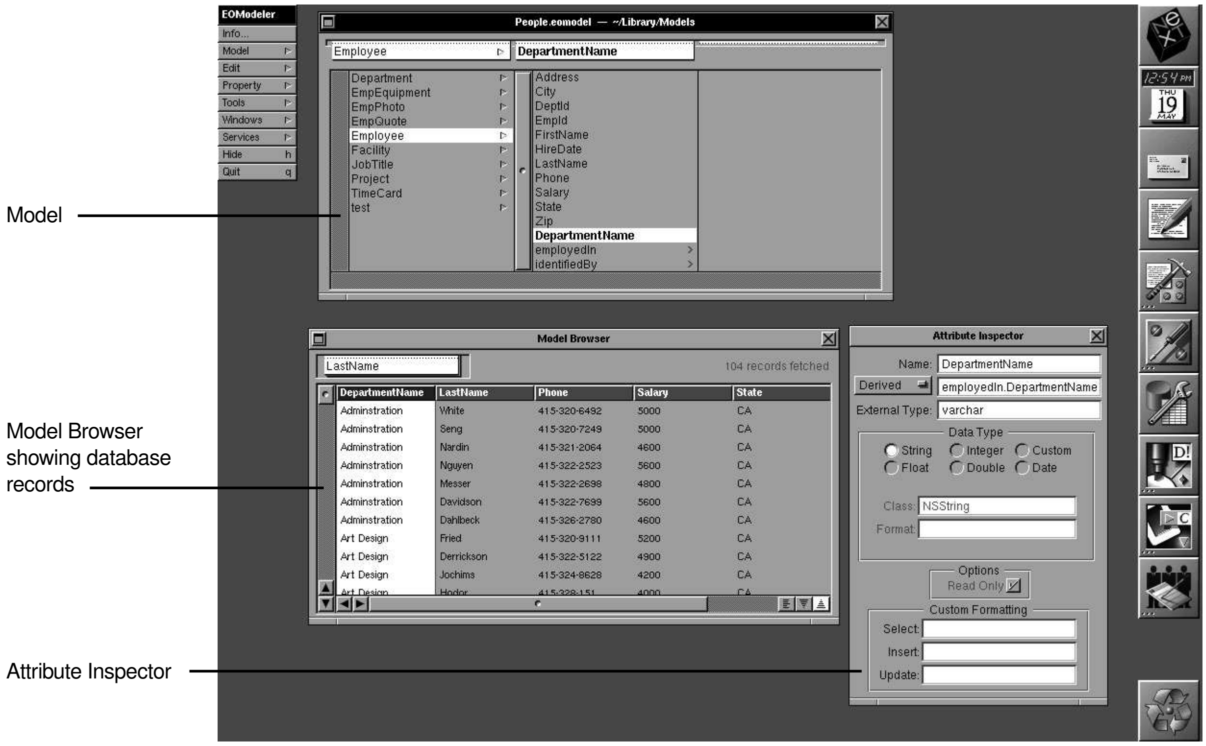



*Figure 4 The EOModeler helps developers map database attributes to enterprise object data*

which can be "wired together" like objects dragged from Interface Builder palettes. Developer-supplied objects and Interface Builder-supplied objects communicate with messages; there's never any code to merge when the user interface is changed. Thus, it's common for applications to be developed in easy-to-manage increments: working in Interface Builder to create the base application, manually adding rudimentary application-specific classes, using the Interface Builder to wire these to UI objects, and fleshing out method implementations one at a time.

### A Flexible, Extensible, Open Framework

"Out of the box" the Enterprise Objects Framework is designed to accommodate a wide range of relational database applications. But where special needs exist, developers can change the behavior of framework objects, can extend the framework with new classes, can substitute their own objects for framework objects, and can choose to incorporate only parts of the framework in their applications. Developers can adapt the framework to accommodate additional relational databases, or even data sources that are not databases at all. New user interface objects can be incorporated into applications and the framework will use them like its own. As always in NEXTSTEP, EO Framework classes can be subclassed and methods can be re-implemented if required.

## II. Technical Perspective

This section answers a number of questions developers may have about the Enterprise Objects Framework:

- What's involved in creating enterprise object classes?
- Given some EO classes, how does application development proceed?
- How does the EO framework work, and what facilities does it provide for developers with special requirements?

Readers with limited interests may wish to read this section selectively.

## Class Development

Well-designed enterprise objects are reusable across many applications. Just as in the database world, EO class design is often done separately from application development. (All objects of one kind are *instances* of the same class; a class definition specifies the variables and methods within the class.)

A class definition describes two kinds of things, variables and methods, which correspond to the data and policies mentioned in the Business Perspective section. The EO Framework creates an object instance from a class definition, from a file called a model, and from the data values in a database row. Each row becomes one instance. The model describes entities, each entity corresponding to a database table. The entity is the key to object creation because it describes the correspondence between table columns and object variables: that the value of the Balance variable, for example, comes from the BALANCE column. The entity also names the Objective C class definition which describes the objects that will be created from the entity. The first job of an EO Framework class developer is to create an entity and tie it to a class definition. The second job is to augment the class definition: to define its methods and perhaps variables that are not obtained from the database.

### Modeling

A model is recorded in an ASCII file which can be created with an editor by a program or with the help of an EO Framework application called the EOModeler.

Figure 4 shows the EOModeler in use. When the EOModeler is started, it logs the user into the database server, reads the database schema, and constructs an initial model. This is classical NEXTSTEP development: the system generates a reasonable first-cut which the developer customizes as necessary.

Models are expressed according to the entity-relationship scheme popular in database textbooks; entity-relationship modeling is independent of any vendor's database. A model consists of entities which correspond to tables or views. An entity consists of attributes (corresponding to columns), and relationships (corresponding to joins between primary keys and foreign keys).

The model window in Figure 4 lists entities in the first column, and the attributes and relationships of the Employee entity in the second. Relationships are distinguished from attributes by > symbols.

Double-clicking on an attribute displays the Attribute Inspector. The Inspector shows the attribute's name and type in the database, its name in the class definition, and so on. Double-clicking on a relationship displays the Relationship Inspector which similarly displays the characteristics of the relationship. Derived attributes (for example, MarketValue = Shares * Price) which are not in the database can be added to an entity, and columns can be flattened across tables (effectively joining columns from two tables).

To verify an entity specification, you can direct the EOModeler to browse it. The EOModeler displays a browser window with columns for each attribute/relationship, and rows showing the corresponding data fetched from the database. The columns in the browser window can be resized and moved, and the rows can be sorted.

Double-clicking on an entity brings up an Entity Inspector. In this Inspector you can specify which attributes make up the primary key and which participate in locking. You also specify the name of the enterprise object class associated with the entity. The default class is a predefined one called EOGenericRecord. An EOGenericRecord object is a simple data-bearing object, essentially a database record cast into a policy-free object. Its only methods are those defined by the EO Framework to get data into the object when it is created or updated, and get data out of the object to display. Although they do not bind policy to data, EOGenericRecords are nonetheless useful. Sometimes data is enough; EOs don't *have* to have policy. And though you may intend to ultimately give your EOs policy, characterizing them initially as EOGenericRecords enables you to verify that data is flowing from the database to the Enterprise Objects to the user interface and back to the database.

### Augmenting a Class Definition

An entity specifies the "data part" of a class of enterprise objects; the class definition specifies the methods and additional variables that are not obtained from the database. You create an enterprise object class definition as you would any class definition, with the Interface Builder and Project Builder.

The EO Framework demands little of EO class developers, the primary requirement being the provision of two data transfer methods. The framework transfers data into an EO (when it creates one or updates it from the user interface) by sending a takeValuesFromDictionary message. When the framework needs data from an EO, it sends a valuesForKeys message. Default implementations of these methods, sufficient for most classes, can be had by reuse: by inheritance from the Object or NSObject classes. The data transfer methods are described in more detail in “Foundation Classes, Memory Management, and Dictionaries.”

An object that responds to takeValuesFromDictionary and valuesForKey is an enterprise object that carries policy-free data. The important methods for an EO class are those that express business polices. It’s up to business analysts and developers to define and implement these methods.

Although the parts of a method that send and receive messages must be written in Objective C, these parts can call functions written in C or C++. If you have a library of C functions that compute the risks of currency fluctuations, or a group of C++ programmers who know how write the same, you can employ the former or the work of the latter in the core of your policy method implementations.

## Building Applications

If you’ve ever built an application with NEXTSTEP, you may want to skip this section; building an application that uses enterprise objects is almost identical to building one that doesn’t. The developer uses the NEXTSTEP Project Builder and Interface Builder to *compose* the application from user interface and enterprise objects and to *connect* those objects so they will send appropriates messages in response to user actions. EO Framework connections differ only slightly from those that NEXTSTEP developers are already familiar with.

Although simple applications can be developed solely by composition and connection, many will also require coding. For example, an application may need a user interface object that is not available on a palette; perhaps a calendar that shows a date as a circled number. Another example is an object that displays a query panel, parses the user’s input, and constructs an qualifier object which specifies the criteria for fetching EOs. (A qualifier is the EO Framework equivalent of an SQL “where” clause.)

The main steps to creating an EO-using application are:

- Launch the Project Builder and direct it to create a new application.
- Launch the Interface Builder and EOModeler.
- Drag entities from the EOModeler to the Interface Builder file window.

  Each entity you drag represents a class of enterprise objects to be used by your application. (In Figure 3, the Department icon represents an entity dragged from EOModeler.)
- Drag UI objects (windows, buttons, and so on) from Interface Builder palettes to application windows, and position, size, and label them.

  The EO Framework adds a new palette to the Interface Builder. The palette contains a UI object called NXTableView which can display (for reading or updating) EO data in tabular form, one row per EO. (Figure 3 shows two NXTableViews.) The palette also provides a UI object called NXImageView for displaying images represented in variety of standard formats. (In Figure 3 an NXImageView is used to display an employee’s photograph.)
- Code the classes that your application needs but are not supplied by the EOModeler or Interface Builder, and drag them into the Interface Builder File window.
- Connect UI objects that display or control enterprise objects to the corresponding entities. For example, to make a button save all EOs, control-drag from the button to the entity icon. The EOController Inspector will appear, showing a list of actions (methods) that the EOController can perform; select the save action, and press the Connect button. (An EOController coordinates the interaction between UI objects and a class of EOs; how this works is described in “The User Interface Layer.”)

  Similarly, to connect a column in a NXTableView to a variable in a class of EOs, control-drag from the entity icon to the column (as is being done in Figure 3). The EOController Inspector will display a list of associations (described in “The User Interface Layer”), each one standing for a variable in the class associated with the entity. Select the variable to be displayed in the column, then press the Connect button.
- Set EOController attributes in the EOController Inspector. These specify how undo is to work, when user changes are applied to EOs, and when EO changes are applied to the database.
- Direct Project Builder to compile and link the application.
- Switch Interface Builder to test mode and exercise the application.

As is usual with NEXTSTEP, the above activities need not be followed linearly. A developer who prefers an incremental style can put together a portion of the user interface, connect its objects to simple policy-free enterprise objects, and test; then refine the user interface and test again; then add methods representing business policy to EOs, test again, and so on.

## Opening the Hood

This section describes aspects of the EO Framework that most developers don't need to know: how the framework is designed and operates, and what facilities it offers to developers with special requirements. For example:

- You can change the default behavior of EO Framework objects by sending them messages. (The framework both creates objects and is composed of them.)
- You can exercise fine or coarse control over database interaction—setting the scope of transactions, even issuing your own SQL commands.
- You can build applications that use only some of the EO Framework objects; for example, if you write an application that has a command-line interface, you have no need for EO Framework user interface objects.
- You can extend the EO Framework to accommodate a type of database that isn't supported by the standard product, or a source of enterprise objects that isn't a database at all.

## Architecture Overview

The architecture of the Enterprise Objects Framework is divided into two connected layers as shown in Figure 5. In each layer are framework objects that, as one looks higher in the diagram, provide a more abstract view of the underlying database. In the top half of the diagram, the view is strictly of enterprise objects; there is no suggestion of an underlying database. The layered design, coupled with the separation of concerns embodied in the objects that comprise each layer, helps developers identify the facilities they want to use or adjust, and—as important—to distinguish them from the ones they do *not* want to affect.

Starting at the top, the user interface layer interacts on one side with user interface objects (text fields, buttons, and so on), and on the other with a data source object. To the user interface layer, a data source can supply, insert, and delete enterprise objects. The data source distributed with the EO Framework is connected by the access layer below it to a relational database. Other data sources developed by third parties can be connected to other kinds of stores. The access layer creates enterprise objects from database rows and updates rows from EOs. An application can interact with the objects in any layer and in any combination.

### Foundation Classes, Memory Management, and Dictionaries

All layers of the EO Framework design use a new class hierarchy called the Foundation classes, a new way of managing memory, and a data packaging mechanism known as dictionaries.

The Foundation classes comprise a versatile collection of value classes, instances of which are often found in enterprise objects. They include:

- NSString which provides the usual string operations and transparently supports unicode and C representations of strings
- NSArray, for arrays of objects
- NSData, uninterpreted byte arrays that can be used for BLOBs (binary large objects) or custom data types
- NSNumber, for C numeric values
- NSDate, for representing dates in many formats and calculating differences

Each class has a mutable and immutable version, the latter's methods exploiting the knowledge that the value cannot change. Copying an immutable NSString, for example, is reduced to returning an Objective C id to the original NSString.

The Foundation's new memory management design is easier to use and less prone to memory leaks than the conventional malloc/free approach. Developers of objects that share an object don't have to establish conventions to ensure that the shared object is properly freed. Yet the technique does not exact the performance penalty of garbage collection.

If a method obtains a reference to an object which it does not need past its current execution, the method does nothing; the object is automatically released when it is no longer needed by any object. If a method needs an object for longer than its current execution (for example, to store the object's id in an instance variable), it sends a retain message to the object. When an object no longer needs a retained object, it sends the object a release message. Retained objects deallocate themselves when each

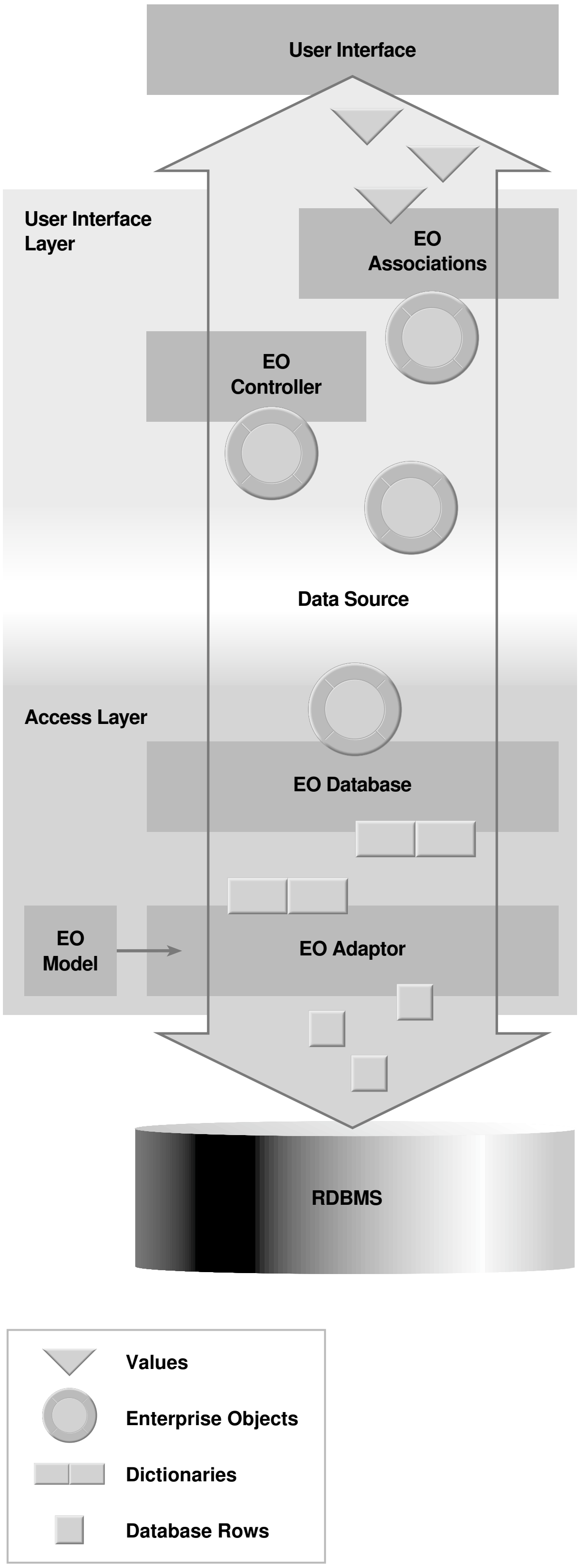



*Figure 5 Simplified EO Framework architecture and dataflow*

retain message they have received has been countered by a release message; that is, when the object is no longer needed by any other object.

Enterprise Object Framework objects use objects called dictionaries to transmit data amongst themselves and to and from EO objects. A dictionary is a collection of key-value pairs. A key is a string that names a value; a value is any object (an id in Objective C).

A dictionary's principal methods are:

- setObjectForKey, which takes a key and a value and inserts them
- objectForKey, which takes a key and returns a value

Dictionaries provide a generic way for EO Framework objects to transmit any kind of named data among themselves and enterprise objects. The fact that a dictionary's entries have names is also valuable in debugging; well-chosen keys can make a dictionary self-describing.

Dictionaries are widely used in the EO Framework: the access layer reads database rows into dictionaries and writes dictionary values into rows; enterprise objects initialize their instance variables from dictionaries, and supply the values of their instance variables in dictionaries. For example, when the framework wants an EO to update an instance variable, it sends the EO a takeValuesFromDictionary message. For each key in the dictionary that matches the name of an instance variable, the EO updates the variable with the associated value. Similarly, when the Framework wants a value from an EO it sends valuesForKeys. The EO responds by returning a dictionary containing the requested key-value pairs. The default implementation of this method allows EOs to return values from storage or to compute them.

### The User Interface Layer

At the heart of the user interface layer is the class called EOController; an application that uses this layer has one EOController for each class of enterprise objects it uses. The EOController takes the place of the "controller" object which is a hand-coded component of most object-oriented applications: it ensures that data displayed on the screen is consistent with the data held in the corresponding enterprise objects. (Those familiar with the model-view-controller paradigm from Smalltalk-80 will find that in the EO Framework, models correspond to enterprise objects; views correspond to UI objects; and controllers correspond to EOControllers.)

An EOController coordinates the following:

- tracking the selection as the user changes it
- distributing updates from user interface objects to EOs
- distributing changes in EOs back to user interface objects

For example, suppose one UI object shows stock price, and another market value which the EO computes as price times shares. If the user changes the stock price, the new price must be transmitted from the UI object to the underlying EO, and the new market value computed by the EO must similarly be transmitted to the other UI object. (If stock price were simultaneously displayed in two UI objects, the same steps would be required to update the second one.)

In addition to coordinating the user interface, an EOController sends messages to a data source to fetch, insert, update, and delete EOs. An EOController and its data source manage the EOs that correspond to one entity.

An EOController maintains the ids of its EOs in an array; one array element corresponds to the current selection in the user interface. Instead of working directly with UI objects, an EOController works with intermediary objects called EOAssociations. There are different kinds of associations for different kinds of UI objects, but all appear identical to a controller. Adding a new user interface class requires creating a corresponding association subclass, but the EOController is untouched; it's designed to accommodate future user interface objects.

An EOAssociation acts as an adaptor between an EOController and a user interface object. One association links one user interface object to one value of the EO class managed by an EOController. "Value" here means as returned by a valuesForKeys message. For example, one association would link a text field labeled "Last Name" to an EO value called LastName. If LastName were associated with a second UI object, it would be by means of a second EOAssociation.

An association maintains two items of state: a key (as in key-value pair) and the id of its UI object. When a UI object changes a value (say, a user changes "Churchill" to "Roosevelt" and presses the Return or Tab key), the following sequence occurs:

- The UI object sends a controlActed message to its association.
- The association sends associationDidEdit to its controller.
- The controller (by default) sends takeValuesFromDictionary to the EO, then updateObject and saveObjects to the data source.
- The controller sends contentsDidChange to its associations.
- Each association sends its key in a valuesForKeys message to the current EO (the one that's selected). The association compares the value returned by the EO with the value it last sent to its UI object; if they are different, it sends an appropriate update message to the UI object, for example, setStringValue to a text field.

Besides interacting with the user interface, an EOController sends fetchObjects, insertObject, updateObject, deleteObject, and saveObjects messages to the data source. It sends fetchObjects when directed to, typically by a button in the user interface. A fetch message can be accompanied by a qualifier object which has the effect of an SQL "where" clause.

By default, an EOController sends the other messages immediately after performing a related operation on an EO, sending a saveObjects message after each insert, update, or delete. (A saveObjects message results in a commitTransaction message in the access layer.) The default approach keeps the EOs and the database consistent. Alternatively, an EOController can be directed to buffer data source operations, sending a batch of them (ending with a saveObjects) when it is sent a saveObjectsToDataSource message. Buffering reduces database traffic but lengthens the interval during which other applications may make conflicting changes to the same data.

Just as it can buffer data source operations, an EOController can queue changes to EOs until it receives a saveToObjects message, then deliver them in a single takeValuesFromDictionary message. Buffering EO changes in this manner is useful for related changes that must be validated as a whole, for example, date changes (an EO needs both the new month and the new day to check their compatibility).

EOControllers also provide applications with an undo facility. When sent an undo message, an EOController reverses all EO changes back to the previous saveObjectsToDataSource message, then notifies the associations to redisplay.

**The Data Source**

A data source is a simple standard interface to a store of enterprise objects. The data source provided with the EO Framework provides an interface to the framework's access layer and ultimately to a relational database. The principal methods a data source must respond to are fetchObjects, insertObject, deleteObject, updateObject, and saveObjects. The simplicity of the interface makes it possible for developers to represent many kinds of stores as data sources; a file or a non-relational database, for example, could be a data source, and thus a source of enterprise objects for the UI layer.

The EODatabaseDataSource, in addition to supporting the data source protocol, insulates the user interface layer from the complexities of database transactions. If sent an insertObject, deleteObject, or updateObject message and no transaction is open, it creates one; all subsequent operations are part of the same transaction until the EODatabaseDataSource receives a saveObjects message, which it responds to by committing the transaction.

Although the EODatabaseDataSource is most often used with the UI layer, it can be used alone as a high-level abstraction that enables developers to retrieve, modify, and save objects with very little code.

**The Access Layer**

Most applications can be written without concern for the facilities provided by the EO Framework access layer. But for those applications that need them, the access layer's objects are open to use by developers. Here are some examples of what can be done with the framework objects that comprise the access layer:

- Use a database-specific feature that's not exposed in the EODatabaseDataSource interface, for example, a stored procedure.
- Send a special SQL expression to a database.
- Throttle fetching to improve user interface responsiveness, that is, fetching a few rows at a time and displaying them, then going back for more rows.
- Change the database update strategy from optimistic to pessimistic locking.

The access layer is divided into a database level and an adaptor level. The objects comprising these levels are similar, differing mainly in the parameters their methods take and return. The adaptor layer transforms database rows into dictionaries and vice versa. The database layer transforms dictionaries into enterprise objects and vice versa.

Three principal kinds of objects are found in the database level:

- EODatabase: a generic connection to a database
- EODatabaseContext: a transaction scope; there can be many contexts per EODatabase; to run a series of operations as a transaction, you create a context, send it beginTransation, send insertObject, fetchObject, and so on, messages to a channel (described next), and finally send commitTransaction to the context.
- EODatabaseChannel: a communication channel to the database; there can be many channels per context; channels provide the capability for multiple concurrently executing operations in a transaction if the database supports them.

Update strategies can be set on a per-transaction basis by sending a setUpdateStrategy message to the EODatabaseContext that's controlling the transaction of interest. Four strategies are available:

1. Update with optimistic locking (the default)
2. Update with pessimistic locking
3. Update with no locking (blind updating)
4. No update (the data is read-only)

The database layer uses the model to transform dictionaries to enterprise objects and enterprise objects to dictionaries. In addition to handling addition to the dictionary-EO transformation, the database layer provides:

- Relationship graphing: As the database level creates enterprise objects, it arranges them in an in-memory graph that corresponds to the relationships defined in the model. Connections between related objects are standard Objective C ids (pointers to objects). As suggested in Figure 2, a database table that represents a company's organization will, when loaded into memory as enterprise objects, look like an organization chart (if one could see objects in memory). Such a graph is the natural representation of related objects, and it's created automatically.

- Relationship faulting: unless directed otherwise, the database level does not fetch related objects until they are referenced; then it fetches them automatically. In other words, database accesses necessary to create related objects in the graph are deferred until required and are then made implicitly. The result is commonly better memory utilization and less database traffic.
- Object uniqueing: If employee 1 is managed by employee 5, and so is employee 2, uniqueing ensures that there is only one instance of employee 5 in memory and that both 1 and 2 point to it. The result is consistency and better performance as well: the database layer won't fetch 5 if it's already in memory.

The adaptor level objects provide generic relational database access, hiding, for example, the dialects of SQL supported by different database vendors. The adaptor level objects are analogous to the database level objects; when you create a database-level object it creates the corresponding adaptor object. Adaptor level messages are analogous to database-level messages except that instead of insertObject, the corresponding message is insertRow, and so on.

Developers who are familiar with a particular relational database's client library can create an adaptor for that database with about the same effort required to write a database application—on the order of a couple of thousand lines of code.

## III. Conclusion

The linkage of relational databases and objects is inevitable. There is no better way to store large amounts of critical data than in a database. The wealth of enterprise information that is already captured in databases cannot be ignored, and should not be ignored: it should be exploited.

As adept as they are at storing data, databases do only that: they have no general-purpose computational facilities. Conversely, procedural code, whether based on 4GLs or compiled languages, leaves out the data. The best way to package business information is to fuse data and policy in objects.

The question is not whether the storage technology of databases will be linked to the software development technology of objects. The question is how.

NeXT's Enterprise Objects Framework is the most comprehensive and versatile approach. It does not change the databases on which existing applications depend. It confers full privileges upon enterprise objects. Its object creation and database access machinery is encapsulated in other objects which have been rigorously tested and never have to be maintained. It supports all facets of development with NEXTSTEP, the most respected object-oriented application environment in the industry. It allows simple applications to be constructed graphically without coding, and more complex ones to be developed incrementally from simple ones. And for special situations, it provides full access to the facilities of its own object-oriented architecture.

NeXT Computer, Inc., 900 Chesapeake Drive, Redwood City, CA 94063
For additional information, call 1-800-TRY-NeXT.

1M5231 6/94