10

Working Across Levels

Each level of the Enterprise Objects Framework maintains its own balance between convenience and flexibility. The highest levelÐthe interface layerÐis the most convenient but in many ways the least flexible, while the lowest levelÐthe access layer's adaptor levelÐrequires you to do more work but offers a lot of power. The level you choose to work with typically has all the functionality you need, but there may be times when you need to ªdownshiftº and use the more powerful features of a lower level (for example, to invoke database-specific functionality or to debug problems appearing at a higher level).

The Framework allows you to do this, as long as you hold to a few simple guidelines. These guidelines are presented in three sections:

- ªEncapsulating Lower-level Workº presents the simplest way to access lower-level features: Leave things at the lower level (such as transaction state) the way you found them when you return to the higher level.
- ªMatching Lower-level Results with Higher-level Expectationsº shows a few cases in which you set up an operation at a lower level and continue it at a higher level.
- ªInforming Higher Levels of Lower-level Actionsº discusses how you can keep the various levels synchronized when you can't clean up, by letting the higher level know what you (or someone else) have done in the course of bypassing it.

A final section, ªUsing Client Libraries,º presents issues you should be aware of on the rare occasions that you need to completely bypass the Framework.

Encapsulating Lower-level Work

The safest way to use lower-level components is to make sure you finish your work at the lower level before returning to the higher level. You should also revert any other changes in the state of the lower-level components or the database itself before you return to the higher level. For example, if you're working with an EOController and need to find a bug by using the controller's adaptor to fetch raw data, you should finish or cancel the fetch and end any transactions you've begun before resuming work with the EOController.

Transaction state is of special concern because the transaction-controlling methodsÐ**beginTransaction**, **commitTransaction**, and **abortTransaction**Ðare only propagated downward. EODatabaseContext objects send these messages to EOAdaptorContext objects, but the EOAdaptorContext objects don't send these messages ªupº to the EODatabaseContext objects that use them. Further, adaptors typically can't track changes in the database server's transaction state. When you use these methods, then, always send them to the higher-level object, even if your actual work involves lower-level objects. If you must leave a state change open, you can use the methods described in ªInforming Higher Levels of Lower-level Actionsº to notify the higher-level objects of

the change.

Matching Lower-level Results with Higher-level Expectations

Sometimes you have to change the state of lower-level objects or of the server to continue work at the higher level. You can do this, provided you acknowledge what state changes have taken place, and what state the higher level expects things to be in. For example, if you're using an EODatabaseDataSource object and need finer control over transactions than its default behavior provides, you can configure it not to begin transactions automaticallyÐbut then you have to explicitly begin and end transactions with the data source's EODatabaseContext. By turning off automatic transactions, you claim responsibility for handling them.

As another example, the Framework allows you to select database records at a very low level and continue at higher levels. EOAdaptorChannel, the class that actually performs fetching, assumes that attributes have been selected in a certain order; as long as you meet this expectation, you can make a selection by nearly any means available, such as sending raw SQL to the server with **evaluateExpression:** or with EOEntity's external query feature. The following sections summarize these procedures.

Retrieving Records with Raw SQL

Normally, when you retrieve records at the adaptor level, you use EOAdaptorChannel's **selectAttributes:describedByQualifier:fetchOrder:lock:** method to select records, and then proceed through a loop using **fetchAttributes:withZone:** to fetch the selected records (see Chapter 8, ªRetrieving Recordsº). **selectAttributes:...** always selects columns in the database in alphabetic order, based on the internal names of the EOAttribute objects requested for selection. **fetchAttributes:withZone:** then assumes that the columns have been selected in this order.

As an alternative to **selectAttributes:...**, you can use **evaluateExpression:** to perform a selection with raw SQL; this allows you to access database-specific features, such as stored procedures. When you do this, you have to make sure the columns specified in the SQL expression are ordered based on the names of the EOAttribute objects they're associated with, and then request only those attributes in your **fetchAttributes:withZone:** messages. Adaptors are responsible for determining whether a SQL expression puts the database server into fetch mode, so you don't have to worry about that; you simply meet the expectations of the EOAdaptorChannel and fetch the records.

For more information, see the descriptions for the relevant methods of the EOAdaptorChannel class in the *Enterprise Objects Framework Reference*.

Retrieving Objects with EOEntity's External Query

When you specify an external query for an entity in EOModeler, or set one programmatically with EOEntity's **setExternalQuery:** method, you're packaging a raw SQL expression with that entity. EODatabaseChannel uses this expression whenever you give it an unrestricted qualifier (one that selects all records for the entity). You typically use an external query to hide columns in the database table or to invoke a stored procedure when the table is accessed. As with **evaluateExpression:**, an external query must select columns in order based on the names of their EOAttribute objects.

You can also retrieve objects with an EOAdaptorChannel using **evaluateExpression:**. Retrieving objects instead of records requires you to specify which enterprise object class to instantiate. This topic is discussed in more detail in the next section.

Informing Higher Levels of Lower-level Actions

Sometimes you need to change the state of a lower-level component and not change it back before returning to the higher level. In these cases you have to inform the higher-level components of any changes that may affect them, such as a change in transaction state. Most of these changes can be made at the higher level to begin with, so you shouldn't have to do this often.

As mentioned in ªEncapsulating Lower-level Work,º adaptors typically can't track the transaction state for their connection to the database. If you need to change the transaction state at a lower level and not change it back, you must inform the higher-level object (EOAdaptorContext or EODatabaseContext) of this change with the transaction-notification methods: **transactionDidBegin**, **transactionDidCommit**, and **transactionDidRollback**. If a SQL expression you send with **evaluateExpression:** changes the transaction state, for example, you should send the appropriate transaction-notification method to the database or adaptor context object associated with the EOAdaptorChannel.

Retrieving Objects with Raw SQL

One notable task that requires you to notify higher levels is performing a selection with raw SQL in order to fetch enterprise objects with an EODatabaseChannel. Normally you use EODatabaseChannel's **selectObjectsDescribedByQualifier:fetchOrder:** method to select objects. The qualifier you specify determines which entity the selection is made for, and thus which enterprise object class is instantiated when data is actually fetched. If you perform the selection at a lower level, you must explicitly tell the EODatabaseChannel which entity it's fetching for with the **setCurrentEntity:** method.

To perform such a dual-level fetch, you follow these basic steps:

1. Get your EODatabaseChannel's EOAdaptorChannel.
2. Perform the selection with EOAdaptorChannel's **evaluateExpression:** method.
3. Send **setCurrentEntity:** to the EODatabaseChannel with the entity you selected in step 2.
4. Use **fetchWithZone:** to fetch objects.

The following code excerpt illustrates this procedure by selecting objects for the Employee entity:

```
EODatabaseChannel *dbChannel;       /* Assume this exists. */
EOEntity *employeeEntity;           /* Assume this exists. */
NSString *aRawSQLExpression;        /* Assume this exists. */
EOAdaptorChannel *adaptorChannel;

[dbChannel beginTransaction];

adaptorChannel = [dbChannel adaptorChannel];

[adaptorChannel evaluateExpression:aRawSQLExpression];
[dbChannel setCurrentEntity:employeeEntity];

while ([dbChannel isFetchInProgress]) {
    id theEmployee = [dbChannel fetchWithZone:NULL];
    /* Process theEmployee. */
}

[dbChannel commitTransaction];
```

Using Client Libraries

In extreme situations, you may find that you have to resort to your database server's client library interface to get something done. If you must do this, you should isolate any code that invokes the client library from code that uses the Enterprise Objects Framework. Any descent into the client library should be as self-contained as possible, leaving no discrepancies between the Framework's view of the database and its true state (especially with regard to transactions). Beyond this warning, little specific advice can be offered.

How you link your database server's client library to your application depends on the form of the client library and on the adaptor you're using. See the documentation for each of these components to determine how you can link the code you need into your application. The Sybase and Oracle adaptors included with the Framework are documented in appendices to the *Enterprise Objects Framework Reference*.