9

Advanced Record List Techniques

The DBRecordList objects that were used in the previous chapter were intentionally simple. This simplicity was concentrated in three areas:

- Their property lists contained only ªnaturalº properties; in other words, properties that were gotten directly from DBEntities objects.
- When the DBRecordLists were told to fetch, *all* corresponding records were retrieved.
- The source entities, as passed as the second argument to the **setProperties:ofSource:**, were all simple DBEntities objects.

The values were retrieved for each record that lies in the table that corresponds to the DBRecordList's source entity. The property list, record range, and source, in this scenario, are unnecessarily restrictive. In reality, all are more flexible:

- The property list can contain objects other than those that are found in a database model. Specifically, you can create your own DBExpression objects and add them to a DBRecordList's property list.
- You can restrict, or ªqualify,º the range of records that are retrieved by employing a DBQualifier object when you tell a DBRecordList to fetch.
- By setting the source of one DBRecordList to a value retrieved by another, you can create a master-detail correspondence between the two objects.

The following sections examine the DBExpression and DBQualifier classes and show how to create a master-detail set-up using DBRecordList objects.

DBExpression

A DBExpression object encapsulates an expression that, when evaluated, represents a column of data that's stored in a DBRecordList. This may sound a lot like the agenda of the database property objects that you get by sending a **getProperties:** message to an entity objectÐthese, too, represent columns of data. The similarity isn't mere coincidence: The DBExpression class adopts the DBProperties protocol, thus a DBExpression object *is* a property. Anywhere a property object is called forÐmost notably in DBRecordList's **setProperties:ofSource:** and the **set/getValue:forProperty:at:** methodsÐyou can use a DBExpression object.

Given that ªnaturalº database properties (in other words, property objects that are gotten through **getProperties:**) and DBExpression objects are similar, why, then should you bother with creating the latter, when you can so easily get the former from the model? Because DBExpression objects can do things that natural properties can't. A DBExpression object can do the following:

- Create a join through a to-one relationship.
- Specify data that's derived from the manipulation of other properties.
- Change the data type of a property.

You can also use a DBExpression to encapsulate, or ªcoverº an existing property. This is mostly a matter of consistency, as it doesn't admit any new functionality.

Traversing To-One Relationships (Joining Tables)

Perhaps the most important facet of a DBExpression object is that it allows you to store, in the same DBRecordList, properties from entities that are related through a to-one relationship. You must use a DBExpression for this, as opposed to simply setting (in a DBRecordList) the desired properties from the related entities. For example, consider the model shown in Figure 77.

Figure_77. A Simple Model

By first asking the Book entity for its properties, and then asking the destination entity of the toAuthor relationship for its properties, you come up with the following lists of property objects (shown here by name and categorized by entity):

Book	Author
title	name
authorID	address
publisher	birthdate
bookID	authorID
toAuthor	

You can't pluck properties from these two lists and set them in the same DBRecordList object because you wouldn't be able to provide a single entity object as the source argument to the **setProperties:ofSource:** method. However, since the Book and Author entities are related through a to-one relationship, it's not unreasonable to expect to be able to use a single DBRecordList to hold records that contain, for example, a book's title and the name of the book's author.

This is where a DBExpression object comes in. To create a DBExpression for the Author.name attribute that's gotten through the toAuthor relationship, you describe the author's name attribute as toAuthor.name rooted at Book. This is performed through the DBExpression's **initForEntity:fromDescription:** method:

```
/* The database object is assumed to exist. */
id bookEntity = [database entityNamed:"Book"];
DBExpression *authorName = [[DBExpression alloc]
                                    initForEntity:bookEntity
                                    fromDescription:"toAuthor.name"];
```

The first argument is an entity object; the second (the object's ªdescriptionº) is a string that gives a model path that's rooted at the first-argument entity, and that ends at a named property in a related entity (see ªThe DBExpressionValues Protocol,º below, for other ways to format this string). The object as a whole represents the Book.toAuthor.name attribute.

The **authorName** DBExpression object and the natural book properties can then be placed in the same DBRecordList:

```
List *aList = [[List alloc] init];
DBRecordList *bookRecordList = [[DBRecordList alloc] init];

/* Add the previously created DBExpression object to the list. */
```

```
[aList addObject:authorName];

/* Add a natural book property to the list. */
[aList addObject:[bookEntity propertyNamed:"title"];
[bookRecordList setProperties:aList ofSource:bookEntity];
```

When you tell the DBRecordList to fetch, the records that are retrieved will contain the title and author of each book that's represented in the database. The correspondence between titles and authors is correctly enforcedÐyou'll get the right author for a given title. Put technically, the source and destination attributes for the toAuthor relationship are automatically applied to create this correspondence.

Nested Relationships

A DBExpression can traverse nested to-one relationships. For example, the following message initializes a DBExpression to represent the address of the union to which the author of a particular book belongs. The model upon which this is based is shown in Figure 78.

```
/* bookEntity is assumed to exist. */
DBExpression *unionName =
                [[DBExpression alloc]
                      initForEntity:bookEntity
                      fromDescription:"toAuthor.toUnion.address"];
```

Figure_78. Nested Relationship Model

The **unionName** object, as defined here, can also be added to the **bookRecordList** object:

```
[aList addObject:unionName];
[aList addObject:authorName];
[aList addObject:[bookEntity propertyNamed:"title"];
[bookRecordList setProperties:aList ofSource:bookEntity];
```

Property Facets

Since the DBExpression class conforms to the DBProperties protocol, you can send a DBExpression object any of the messages that are defined by that protocol. In particular, you can ask a DBExpression for its entity, name, property type, and so on. The responses that a relationship-traversing DBExpression gives to these messages are described below:

Method	Response
name	The object's name is its description. For example, the name of the **authorName** DBExpression is toAuthor.name. There's an important distinction between a natural property's name and that of a DBExpression object: The natural property's name is global, the DBExpression's isn't. Because of this, you can't use the DBEntities method **propertyNamed:** to find a DBExpression. If you asked the **bookEntity** object for the property named toAuthor.name, the **propertyNamed:** method would return **nil**.
entity	The entity is that at which the DBExpression is rooted; in other words, it's the object that you used as the first argument to the **initForEntity:fromDescription:** method.

The responses to the other messages are gotten by forwarding the message to the ªforeignº

propertyÐin other words, the object that naturally represents the property in the related entity. However, some of these responses can't be taken literally, as described below:

Method	Response
propertyType	The property type object of a relationship-traversing DBExpression is (accurately) retrieved from the foreign property. For example, the property type of the **authorName** object used above is the same as the property type of the Author.name attribute.
isSingular	The response to this message is also accurately retrieved from the foreign property.
isReadOnly	The response to this message doesn't matter: You can't alter the values that are held for a relationship-traversing DBExpression. Such objects are, in practice, read-only. The reason for this is explained in the next section.
isKey	Here, too, the response doesn't matter. A relationship-traversing DBExpression *can't* represent a primary key of the root entity since it doesn't represent a property that naturally resides in that entity. (Where the relationship is reflexive the discussion tends to the philosophical. We'll leave it to the theorists.) The important point here is that the automatic key-adding mechanism defined by DBRecordList's **setProperties:ofSource:** method isn't fooled by the response: If you add a related attribute that's declared to be a primary key (for its natural entity), it won't be added to the DBRecordList's list of key properties.

Specifying Derived Data

The second use of DBExpression objects lets you create properties that represent derived data, or values that are created by manipulating the values for other properties. For example, consider a simple Box entity that contains the attributes height, width, and depth. Furthermore, for each box record, you want to compute the volume and store it as a separate property. To do so you describe a DBExpression thus:

```
DBExpression *volume =
                  [[DBExpression alloc]
                        initForEntity:boxEntity
                        fromDescription:"height * width * depth"];
```

As shown here, the description string isn't just a simple model path that's rooted at the entity argument; the format of the description shown here is explained in the following sections.

The DBExpression that you've created can be set in a DBRecordList's property list (along with other properties that are rooted at **boxEntity**). When the DBRecordList fetches data, the volume computation is automatically performed using the values for the height, width, and depth properties of each record. You can then use the **volume** DBExpression object as a property to retrieve the volume value of individual records:

```
/*    boxRecordList is the DBRecordList that fetched box records.
   val is a DBValue that's assumed to exist. */
[boxRecordList getValue:val forProperty:volume at:n]
```

Description String Evaluation

To understand how a derived data DBExpression works, you need to understand the mechanics by which the object's description string is evaluated. (The outline given below actually applies to all DBExpression objects, although its complexity is only discovered when you create derived data.)

When you fetch data into a DBRecordList that contains a DBExpression, the string that describes the DBExpression is parsed and evaluated to form a statement in the server's query language:

- Parsing a description string simply means that the individual components are identified and separated; for example, the components used in the previous example are ªheightº, ª*º, ªwidthº, ª*º, and ªdepthº.

- Next, the substitution symbols (if any) are replaced by the corresponding arguments. This is explained in the next section (the DBExpression used in the example above contained no substitution symbols).

- The components are then evaluated by the Database Kit. It looks at a component, decides if it recognizes it as an object and, if it does, sends that object an **expressionValue** or **stringValue** message. The latter is sent if the object doesn't respond to the former; the significance of these messages is examined later. This message retrieves, from the receiving object, an expression or symbol that's assumed to be valid in the server's query language. For example, the expression value of a property object that's retrieved from a database model is the internal name of the property as set through the DBModeler application. Anything the Database Kit doesn't recognize is left unevaluated. In the example above, the Database Kit would recognize and evaluate the property names, but not the ª*º symbols.

- Finally, the string is sent to the adaptor, which applies the parsed and Kit-evaluated description to each record that's involved in the transaction. This creates a value that's sent back to your application and stored (in the DBRecordList) for the DBExpression object that started all of this.

During its evaluation, the Database Kit performs a second task: It decides whether the DBExpression represents derived data, as opposed to a traversed relationship or, as explained later, whether it's being used to type-cast or cover a property. The rule for determining the intent of a DBExpression is fairly simple: If the description string contains anything other than references to property objects, the DBExpression is assumed to represent derived data.

This determination is important because it regulates the data type of the DBExpression object and determines whether values that are retrieved for the object are read-only:

- The data type of DBExpression object that represents derived data is always a string, even if the computation naturally results in numeric data.

- Derived-data DBExpressions are read-only. You can't write derived-data back to the server. (This should be obvious since the DBExpression, in this case, doesn't represent an actual ªcolumnº of data on the server.)

Description String Format

The format of a DBExpression's description argument is similar to **printf**-type statements: It consists of a quoted string containing the property names, operators, and substitution symbols needed to construct the desired expression, followed by values to be substituted for the symbols. The following substitution symbols may occur within the description string:

Symbol	Value
%s	A constant string (**const char ***).
%p	A string that names one of the entity's properties.
%d	An **int**.
%f	A **double** or **float**.
%@	An object that conforms to the DBExpressionValues protocol or that implements the **stringValue** method (see below).
%%	No valueÐthis passes a single `%' literally.

For example, the following description creates a DBExpression that adds 2.5 to the value of the

height property:

```
float higher = 2.5;
DBExpression *newHeight =
                [[DBExpression alloc]
                      initForEntity:boxEntity
                      fromDescription:"%p + %f", "height", higher];
```

After parsing and evaluating, the Database Kit converts this string into the expression

```
HEIGHT + 2.5
```

(where ªHEIGHTº is assumed to be the private name of the height property) and sends the expression to the adaptor.

Actual values, whether numbers or strings, are passed literally to the adaptor. To protect a string values from being evaluated by the Database Kit, the value that replaces the ª%sº substitution symbol is placed in single quotes. For example, the evaluation of the DBExpression defined as

```
float higher = 2.5;
DBExpression *calcHeight =
                [[DBExpression alloc]
                      initForEntity:boxEntity
                      fromDescription:"%p %s %f %s %p + %f",
                      "height", " plus ", higher, " equals ",
                      "height", higher];
```

produces an expression that appears as

```
HEIGHT ' plus ' 2.5 ' equals ' HEIGHT + 2.5
```

This expression is evaluated for each record that's fetched. Given a record that contains the value 4.3 for its height property, the value at **calcHeight** would be

```
4.3 plus 2.5 equals 6.8
```

(Since derived data is always cast as a string, the prosaic nature of the value doesn't pose a problem.)

You can insert string values directly into the description string, but you must quote them yourself, as shown in the following example.

```
DBExpression *newHeight =
             [[DBExpression alloc]
                initForEntity:boxEntity
                fromDescription:"%p ' plus ' %f ' equals ' %p + %f",
                "height", higher, "height", higher];
```

The DBExpressionValues Protocol

The DBExpressionValues protocol, which declares the **expressionValue** method, is conformed to by DBValue and DBExpression objects, and by the entity and property objects that are retrieved from a database model. Any of these objects, therefore, can be used in a DBExpression object's description as the replacement value for the ª%@º symbol.

There are two uses of DBExpressionValues-conforming objects in the description string that deserve special mention:

- **Nested DBExpressions**. DBExpression is one of the classes that conforms to the DBExpressionValues protocol; this means that you can created ªnestedº DBExpression objects. For example, an even higher height property can be derived by nesting the previous example's **newHeight** object in another DBExpression:

```
float evenHigher = 3.6;
DBExpression *higherHeight =
```

```
                    [[DBExpression alloc]
                     initForEntity:boxEntity
                     fromDescription:"%@ + %f", newHeight, evenHigher]
```

- **Constructing a relationship-traversal string.** By relying on the DBExpressionValues conformity of database entity and property objects, you can programmatically construct a string that traverses a relationship. Below, the ªtoAuthor.nameº string is constructed through substitution. In its evaluation of the string, the Database Kit understands the embedded period to denote a model path delimiter; thus, the example *doesn't* create derived data (and so the DBExpression isn't declared read-only, and its data type isn't, of necessity, string):

```
/* bookEntity is assumed to exist. */
id toAuthorRel = [bookEntity propertyNamed:"toAuthor"];

/* Get the desired attribute from the relationship's destination
   entity. */
id authorName = [[toAuthorRel propertyValue] propertyNamed:"name"];

/* Now construct a relationship-traversing DBExpression from these
   objects. */
DBExpression *authorName =
                [[DBExpression alloc]
                          initForEntity:bookEntity
                          fromDescription:"%@.%@",
                          toAuthorRel, authorName];
```

Using Other Objects in your DBExpression

If the object that's substituted for the ªº%@º symbol doesn't conform to the DBExpressionValues protocol, then it's evaluated by the Database Kit through the **stringValue** method. This is a handy way to get the value from a user interface object, such as a Slider or a TextField, into a DBExpression. Here, the value of a Slider is added to the height property:

```
/* aSlider is assumed to exist as part of the user interface. */
DBExpression *newestHeight =
                  [[DBExpression alloc]
                        initForEntity:boxEntity
                        fromDescription:"%p + %@", "height", aSlider];
```

Of course, Sliders don't naturally yield string values, so this example may seem a bit odd. However, when the **newestHeight** object is evaluated, the value of **aSlider**, converted to a string by **stringValue**, is placed (unquoted) in the statement that's passed to the adaptor. In other words, the query language statement will look the same whether the Slider's value is entered into the description as a string or as a number. Thus, the above example produces the same expression as the following:

```
DBExpression *newestHeight =
                  [[DBExpression alloc]
                        initForEntity:boxEntity
                        fromDescription:"%p + %f",
                        "height", [aSlider floatValue];
```

The expression itself for either of these examples, given a Slider value of, say, 2.5, would be:

```
HEIGHT + 2.5
```

However, don't let the Database Kit's invocation of **stringValue** lead you to think that you can get the same result by invoking the method yourself and placing the retrieved value in the description as a string. If you do this, the value will appear quoted in the expression, causing the adaptor to think that it really *is* a string. To demonstrate this, we rewrite the preceding example to invoke **stringValue** directly:

```
DBExpression *newestHeight =
```

```
                    [[DBExpression alloc]
                          initForEntity:boxEntity
                          fromDescription:"%p + %s",
                          "height", [aSlider stringValue];
```

This produces the following expression:

```
HEIGHT + '2.5'
```

Conversely, you may really want to pass a string value as a quoted string, in which case this last example is the way to do it.

Casting a Property's Data Type

The property objects that you retrieve from a database model have a data type that can't be changed. If, for example, the height property represents **float** values, it will always represent **float** valuesÐyou can't tell the property to represent **double**s or **int**s. While you can't change a property's data type, you can recast it by placing the property in a DBExpression. This is done through the **initForEntity:fromName:usingType:** method:

```
DBExpression *doubleHeight =
                    [[DBExpression alloc]
                              initForEntity:boxEntity
                              fromName:"height"
                              usingType:"d"];
```

The first two arguments identify, by entity object and property name, the property that you want to cast. Note well that the second argument is always a property name (possibly of a related entity, as formed through the ª*relationship.attribute*º format)Ðit can't be constructed in the fashion of a description string. Furthermore, the string must identify a naturally occurring model path rooted at the given entityÐyou can't use this method to recast the data type of a DBExpression object.

The final argument uses the data type string cconventiononvention that should be familiar from the Chapter 6 discussion of DBTypes objects.

Covering a Property

Finally, you can use DBExpression objects to represent natural properties. For example, the Box.height property can be covered by a DBExpression object thus:

```
DBExpression *boxHeight =
                    [[DBExpression alloc]
                              initForEntity:boxEntity
                              fromDescription:"height"];
```

This use of a DBExpression doesn't provide any new functionalityÐthere's no significant difference between the covered property and the DBExpression that covers it. However, you may find it beneficially consistent to store, in a DBRecordList, *only* DBExpression objects. Having created DBExpressions for traversing relationships, to represent derived data, and to recast property types, you may want to go ahead and cover all the other properties that you're interested in.

Notice that the description string in the example above doesn't contain anything that the Database Kit can't evaluate. Thus, covering a property won't recast the object's data type to string, or automatically declare it as read-only.

DBQualifier

DBQualifier objects let you define the range of records that are retrieved during a fetch based on a formula that compares, at each record, the values for various properties. For example, let's say you want to retrieve employee records for all employees that make more than $5000 a month. The DBQualifier object that represents this restriction would look like this:

```
id emp = [db entityNamed:"Employee"];
DBQualifier *fiveGEmps =
                [[DBQualifier alloc] initForEntity:emp
                      fromDescription:"salary > 5000"];
```

Any properties that are named in the description string are assumed to be rooted at the entity given by the first argument. In the example, salary must be rooted at the entity represented by **emp**.

Description Format

A DBQualifier's description string obeys the same formatting rules laid out for DBExpression objects. The DBQualifier in the above example could, therefore, be initialized thus:

```
id emp = [db entityNamed:"Employee"];
id salary = [empEntity propertyNamed:"salary"];
int bucksAMonth = 5000;

DBQualifier *fiveGEmps =
                [[DBQualifier alloc] initForEntity:emp
                           fromDescription:"%@ > %d",
                           salary, bucksAMonth];
```

Multiple Qualifications

A DBQualifier's description can contain more than one qualification, provided the query language defines symbols that perform logical operations such as union and intersection. Here, the ªORº symbol is presumed to create the union of two qualification expressions; the parentheses, used for grouping, must also be defined by the query language:

```
id emp = [db entityNamed:"Employee"];
id salary = [emp propertyNamed:"salary"];
id commission = [emp propertyNamed:"commission"];
int bucksAMonth = 5000;
float commissionRate = 25.0;

DBQualifier *fiveGEmps =
                [[DBQualifier alloc] initForEntity:emp
                      fromDescription:"(%@ > %d) OR (%@ > %f)",
                      salary, bucksAMonth,
                      commission, commissionRate];
```

By applying this DBQualifier to a fetch, you would retrieve the records of those employees that make more than $5000 a month or that have a commission exceeding 25%.

You can build a multiple-qualification DBQualifier object through successive invocations of the **addDescription:** method. The description string that's passed as the argument to this method is added to the description that it already holds. The following example builds a DBQualifier from separate descriptions (using the variables declared in the previous examples):

```
/* We start with an empty description. */
DBQualifier *fiveGEmps =
                [[DBQualifier alloc]
                      initForEntity:empEntity
                      fromDescription:""];
```

```
/* Now add the desired qualifiers. */
[fiveGEmps addDescription:"(%@ > %d)", salary, bucksAMonth];
[fiveGEmps addDescription:" OR "];
[fiveGEmps addDescription:"(%@ > %d)", commission, commissionRate];
```

As implied by the example, the concatenation of descriptions doesn't automatically interpose a logical operator or even whitespace, so you must take care when building a DBQualifier in this manner.

Applying a DBQualifier

There's only one thing you can do with a DBQualifier: Use it as the argument to a fetch message, as shown below:

```
DBRecordList *employeeList = [[DBRecordList alloc] init];
List *propList = [[List alloc] init];
id emp = [db entityNamed:"Employee"];
DBQualifier *fiveGEmps =
                [[DBQualifier alloc] initForEntity:emp
                      fromDescription:"salary > 5000"];

/*    Configure the DBRecordList. */
[propList addObject:[emp propertyNamed:"name"];
[propList addObject:[emp propertyNamed:"salary"];
[employeeList setProperties:propList ofSource:emp];

/* Fetch data while applying the DBQualifier. */
[employeeList fetchUsingQualifier:fiveGEmps];
```

Given an employee table on the server, as depicted in Figure 79, the fetch would retrieve only the records shown in Figure 80. As always, the primary key (empID) is fetched automatically.

Figure_79. The Server's Employee Table

Figure_80. The Result of the Qualified Fetch

In regard to the correspondence between a DBQualifier and a DBRecordList, you should note the following rules:

- For a qualified fetch to work, the DBQualifier's entity must match the entity of the DBRecordList that's doing the fetching. If they don't match, the fetch will fail.
- The property objects that are used in a DBQualifier's description needn't be present in the DBRecordList's list of properties. This is shown in the examples in the following sections.

Qualifying Across a Relationship

The properties that are used in a DBQualifier needn't all be directly contained in the object's root entity: A DBQualifier can compare values that are gotten through to-one relationships. To do this, you simply identify the related property in the object's description string as you would in the description of a DBExpression. Specifically, you can identify the property directly:

[a]*toEntity.attribute* > *someValue*[o]

or through substitution:

ª%@ > *someValue*º, *relatedPropertyObject*

In the example below, a DBQualifier is created that compares each employee's department name to ªSALESº, where the department name is gotten through the toDepartment relationship (and notice that the DBRecordList that applies the DBQualifier doesn't itself include the department name property):

```
DBRecordList *employeeList = [[DBRecordList alloc] init];
List *propList = [[List alloc] init];
id emp = [db entityNamed:"Employee"];

/*    Create an object for the "toDepartment.name" attribute. */
id deptName =
   [[[emp propertyNamed:"toDepartment"]
                              propertyValue]
                                   propertyNamed:"name"];

/* The qualification compares the related attribute to the given
   string value. */
DBQualifier *salesFolk =
                  [[DBQualifier alloc] initForEntity:emp
                        fromDescription:"%@ LIKE 'SALES'",
                        deptName];

/*    Configure the DBRecordList. */
[propList addObject:[emp propertyNamed:"name"];
[propList addObject:[emp propertyNamed:"salary"];
[employeeList setProperties:propList ofSource:emp];

/* Fetch data while applying the DBQualifier. */
[employeeList fetchUsingQualifier:salesFolk];
```

To illustrate the result, first consider the model and server tables built from it as depicted in Figure 81.

Figure_81. A Model and Tables

The DBQualifier causes the fetch operation to join the two tables, and then select data from the records that have the requested department name value. The resulting table (as stored in the DBRecordList) is shown in Figure 82.

Figure_82. Result of the Relationship-traversing DBQualifier

DBExpressions in a DBQualifier

A DBQualifier can use DBExpression objects in its description string. When a fetch is performed, the DBExpression's description is evaluated first, and then the DBQualifier is applied. For example, recall the **volume** DBExpression that was created earlier:

```
DBExpression *volume =
                  [[DBExpression alloc]
                        initForEntity:boxEntity
                        fromDescription:"height * width * depth"];
```

The object is placed in a DBQualifier:

```
DBQualifier *roomy =
                [[DBQualifier alloc]
                      initForEntity:boxEntity
                      fromDescription:"%@ > 150.0", volume];
```

During a fetch that's qualified by **roomy**, the enclosed **volume** DBExpression is evaluated and its result is used in **roomy**'s qualification formula. The entities of all three objectsÐthe DBExpression, DBQualifier, and the fetching DBRecordList (which isn't shown in the example)Ðmust be the same.

Master-Detail Record Tables

To create a master-detail set-up you need two DBRecordList objectsÐone to act as master and the other as detailÐand a DBValue object. Assembly proceeds thus:

1. Set the master DBRecordList's property list. One of the properties that it must include, for it to be a master, is a relationship.

2. Prepare the detail property list, but don't set it yet. In other words, create and fill the List object that you'll pass to **setProperties:ofSource:**, but don't invoke the method. The properties that you add to the List object must be gotten from the entity that's retrieved by sending **propertyValue** to the master's relationship. Recall that sending this message retrieves the relationship's destination entity; thus, the detail property list will contain properties of an entity that's related to the master's entity through a to-many relationship.

3. Tell the master DBRecordList to fetch.

4. Get a value for the master's relationship through the **getValue:forProperty:at:** method.

5. Send **setProperties:ofSource:** to the detail DBRecordList object, passing the List object that was prepared in step 2 as the property list argument, and the DBValue that was gotten in step 4 as the source.

6. Tell the detail DBRecordList to fetch.

7. Repeat steps 4-6 for each set of detail records that you want you to store in the detail DBRecordList. Obviously, if you want to store each detail in a separate DBRecordList you'll need to create more DBRecordList objects, but otherwise the steps are the same.

For example, consider a model for the Department and Employee entities shown in Figure 83.

Figure_83. A Model for a Master-Detail Set-up

In real terms, more than one employee may work in a department. To store information for all employees that work in a given department, you need a master-detail configuration: The master DBRecordList will contain Department propertiesÐmost significantly, it must contain the toEmployee relationshipÐand the detail DBRecordList will contain properties of Employee.

Demonstrated in code, the configuration looks like this:

```
/* Create two DBRecordLists, a List object for setting properties,
   and a DBValue to be used after the master has fetched.  */
DBRecordList *masterDept = [[DBRecordList alloc] init];
DBRecordList *detailEmp = [[DBRecordList alloc] init];
List *props = [[List alloc] init];
DBValue *toEmpVal = [[DBValue alloc] init];
```

```
/* Create objects to store the desired entity and properties.
   We'll assume that db, a DBDatabase object, exists.*/
id dept = [db entityNamed:"Department"];
id deptName = [dept propertyNamed:"name"];
id toEmp = [dept propertyNamed:"toEmployee"];

/* Configure the master DBRecordList. */
[props addObject:deptName];
[props addObject:toEmp];
[masterDept setProperties:props ofSource:nil];

/* Empty the List object and reuse it to hold the detail properties. */
[props empty];
[props addObject:[[toEmp propertyValue] propertyNamed:"name"];
[props addObject:[[toEmp propertyValue] propertyNamed:"salary"];

/* Fetch the master records. */
[masterDept fetchUsingQualifier:nil];

/* Get the first record's value for the relationship. */
[masterDept getValue:toEmpVal forProperty:toEmp at:0];

/* Set the detail DBRecordList and fetch. */
[detailEmp setProperties:props ofSource:toEmpVal];
[detailEmp fetchUsingQualifier:nil];
```

To get the employee records for another department (in other words, for the department that's represented by another record in the master DBRecordList), you simply move the cursor, ask for the value of **toEmp** in the current record, and reset the properties in the detail DBRecordList and tell it to fetch.