# NSPersistentDocument Core Data Tutorial

**Data Management**

**2009-02-04**

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



Simultaneously published in the United States and Canada.

# Contents

# Figures, Tables, and Listings

# Introduction to NSPersistentDocument Core Data Tutorial

This tutorial takes you through the steps of building a simple Core Data–based application using `NSPersistentDocument` and Cocoa bindings. `NSPersistentDocument` is a subclass of `NSDocument` that integrates with the Core Data framework. You will find this tutorial useful if you're using the Core Data framework to create a document-based application.

The task goal of this tutorial is to create a document-based application that allows a user to display and modify information about a department, employees in the department, and managerial relationships between the employees.

**Important:** This tutorial is targeted at Mac OS X v10.5 and later. If you're targeting Mac OS X v10.4, see NSPersistentDocument Core Data Tutorial for Mac OS X v10.4.

## Who Should Read This Document

You should read this document to gain an understanding of how to create a Core Data document-based application using `NSPersistentDocument` and Cocoa bindings. Among other concepts, you will learn how to create the project, how to customize the creation of a document, and how to localize error messages.

**Important:** To complete this tutorial, you must already be familiar with basic Cocoa development practices and with Cocoa bindings. This document does not repeat fundamental Cocoa programming concepts and does not provide explicit instructions for common operations (such as using Interface Builder, including establishing bindings between user interface objects and controllers). You should, for example, ensure that you understand the material presented in *Cocoa Application Tutorial Using Bindings*.

## Organization of This Document

"Overview of the Tutorial" (page 11) describes the application you will create, and the task constraints.

"Creating the Project, Model, and Interface" (page 15) describes how you create a Core Data document-based project in Xcode, and how you create the data model and how you can use it to automatically create a default user interface.

"Creating a Custom Employee Class" (page 27) describes how to implement a custom class for an entity.

"Adding a Department Object" (page 33) describes how you add a Department object to the document, and configure the user interface appropriately.

"Copy and Paste" (page 41) describes one way you can support copy and paste in a Core Data application.

"Localizing and Customizing Model Property Names and Error Messages" (page 49) describes how you can localize property names and customize alert panels.

"Document Metadata" (page 55) describes how you can add metadata to your document that Spotlight can extract—it also describes how you write the Spotlight importer.

"A Sheet for Creating a New Employee" (page 61) describes how you can use a sheet for data entry.

# See Also

*Core Data Programming Guide* describes functionality provided by the Core Data framework from a high-level overview to in-depth descriptions.

*Core Data Utility Tutorial* takes you through the steps of building a command-line utility that uses Core Data. *You are strongly encouraged to complete the low-level tutorial before following this tutorial.*

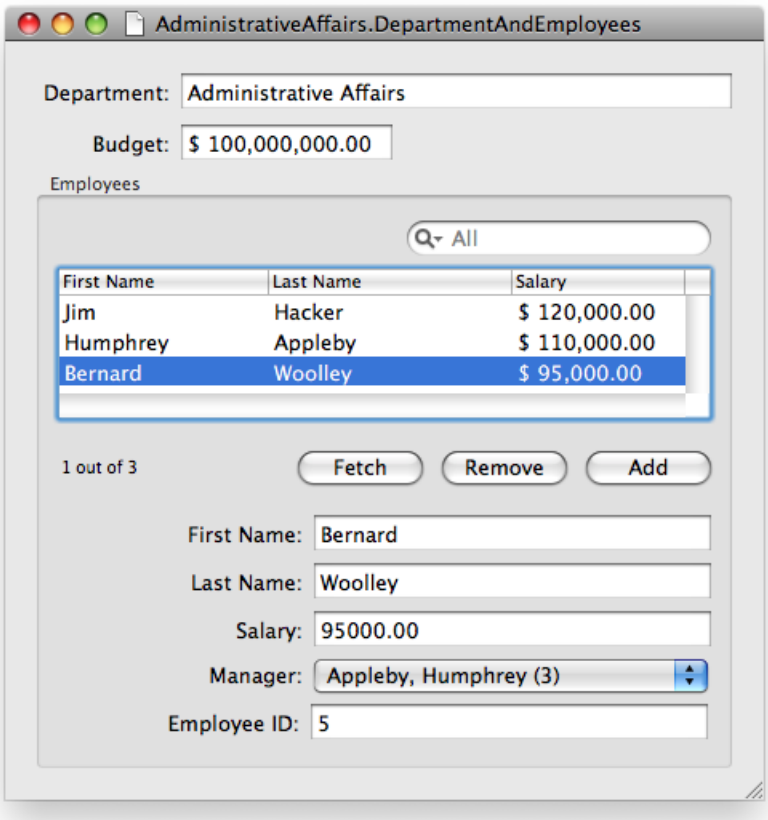
# Overview of the Tutorial

The task goal of this tutorial is to create a document-based application that allows a user to display and modify information about a department, employees in the department, and managerial relationships between the employees. Each document (file) contains information about a single department and the employees associated with it. The application allows the user to save a document as a file and then reopen the file, and supports undo and redo.

## The User Interface

The final user interface may look like that shown in Figure 1-1.

**Figure 1-1** Final User Interface



Note that the emphasis in this tutorial is on functionality. Little attempt is made to refine the user interface as would be appropriate in a shipping application. Moreover, the problem domain is chosen for consistency with other documentation and for clarity rather than for realism.

# Tutorial Steps

This document is intended to give a high-level view of creating a functional application with little code. Although some explanation is given of what happens behind the scenes, this document does not give an in-depth analysis of the Core Data infrastructure.

1. "Creating the Project, Model, and Interface" (page 15)

   The first step illustrates the creation of a Core Data document-based project in Xcode. The major initial requirement is to create the data model, which you can use to automatically create a default user interface. Between Core Data and Cocoa bindings, this first step actually creates a fully functional application that meets most of the task goals without the need to write any code!

2. "Creating a Custom Employee Class" (page 27)

   This part shows you how to implement a custom class for the Employee entity.

3. "Adding a Department Object" (page 33)

   The first step deals only with employees. The next step is to add a Department object to the document and configure the user interface appropriately. One issue that arises here is ensuring the uniqueness of the department. You want to ensure that only one department is created per document.

4. "Copy and Paste" (page 41)

   This part of the tutorial illustrates one approach to supporting copy and paste in a Core Data application.

5. "Localizing and Customizing Model Property Names and Error Messages" (page 49)

   Sometimes the user generates more than one error in a single operation. Core Data provides a rich infrastructure for specifying constraints on data values and error checking, so it's fairly easy to catch these mistakes, but it can be challenging to present the error messages in a useful way. `NSDocument` provides an API to allow alert panels to be customized, so you can tailor customize the error presented to the user to make it as informative as possible.

6. "Document Metadata" (page 55)

   Spotlight provides users with a means of searching for files quickly and easily. To support this, you need to associate metadata with your documents. Core Data makes it easy to do this, and to write the necessary importer.

7. "A Sheet for Creating a New Employee" (page 61)

   This part describes how you can use a sheet for data entry.

# NSPersistentDocument Limitations

Although this tutorial does not aim to cover all the possible features you might implement in an application, some functionality is omitted due to limitations in `NSPersistentDocument` itself. Because of the way Core Data operates, it is not possible to easily support autosaving in an `NSPersistentDocument`-based application. Core Data cannot save to a store and maintain the same changed state in a managed object context, all while keeping an unsaved stack around as the current document. For similar reasons, `NSPersistentDocument` does not support Save To operations.

# Creating the Project, Model, and Interface

This part of the tutorial guides you through building the Department and Employees application and in the process teaches you the steps essential to building a Cocoa application using Core Data and `NSPersistentDocument`.

## Create a New Project

Core Data is integrated into the Cocoa framework, so any Cocoa application can use it. The Employees and Departments application you'll build is a Core Data document-based application. Follow these steps to create the initial project:

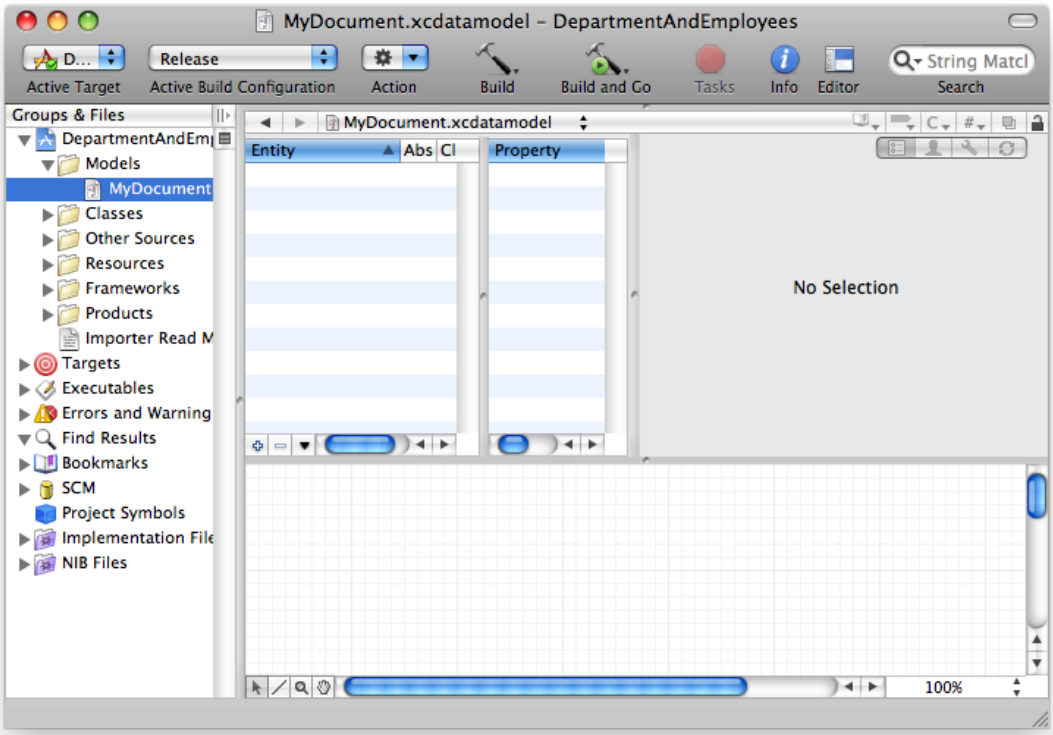
1. Choose New Project from the File menu.
2. Select Core Data Document-based Application with Spotlight Importer in Xcode's project Assistant window, and press Next.
3. Enter the project name (for example, "DepartmentAndEmployees") and a destination folder for the project.
4. This project uses garbage collection (see *Garbage Collection Programming Guide*), so in the project build settings make garbage collection required (see Garbage Collection for Cocoa Essentials).

## Create the Data Model

When you start to develop a Cocoa application, you often begin by prototyping the application's user interface. If you use Core Data, however, your first step is typically to create a data model (or schema) that describes the entities that you will use in your application. Core Data uses the model to help you quickly build full-featured prototypes.

Xcode has a data modeling tool that you use to define the schema for your application, as shown in Figure 2-1. The tool itself is described in *Xcode Tools for Core Data*. If you do not know how to use the tool, consult *Creating a Managed Object Model with Xcode* , which describes how to create a data model.

**Figure 2-1** Data modeling tool




After creating the project, open `MyDocument.xcdatamodel` in the modeling tool by selecting its icon in the project Models folder. Use the tool to define two entities, Employee and Department as specified in Table 2-1 through Table 2-4. The diagram for the finished model should look like that shown in Figure 2-2.

There are other aspects of the schema that are not described in the tables:

- The class for both entities is `NSManagedObject`; neither entity is abstract.
- None of the properties is transient.
- Employee has a reciprocal relationship—a relationship to itself—that defines the manager-reports relationship.
- For this part of the tutorial, the Employee's department relationship is optional.
- The delete rule for all relationships is Nullify.
- The minimum value for the Employee salary attribute is 0.

**Figure 2-2** Diagram for completed data model

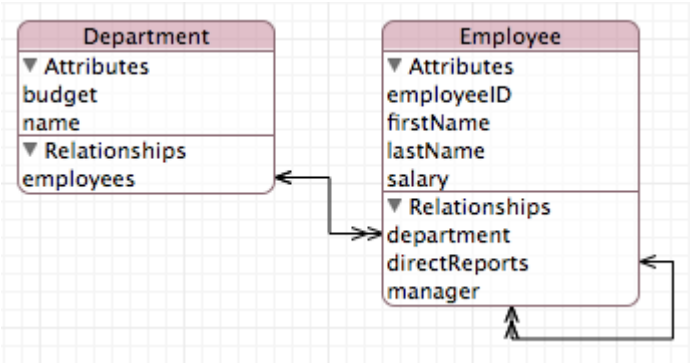



**Table 2-1** Attributes for the Employee entity

| Name | Type | Optional | Default value | Indexed |
|---|---|---|---|---|
| firstName | String | NO | First | Yes |
| lastName | String | NO | Last | Yes |
| employeeID | int 32 | YES | 0 | No |
| salary | Decimal | YES | 0 | No |

**Table 2-2** Relationships for the Employee entity

| Name | Destination | To-many | Optional | Inverse |
|---|---|---|---|---|
| department | Department | NO | YES | employees |
| manager | Employee | NO | YES | directReports |
| directReports | Employee | YES | YES | manager |

**Table 2-3** Attributes for the Department entity

| Name | Type | Optional | Default value | Indexed |
|---|---|---|---|---|
| name | String | NO | Department | No |
| budget | Decimal | YES | 0 | No |

**Table 2-4** Relationships for the Department entity

| Name | Destination | To-many | Optional | Inverse |
|---|---|---|---|---|
| employees | Employee | YES | YES | department |

Note that the relationships are all modeled in both directions. It is important that you do this to ensure the integrity of the object graph (so that Core Data can keep the relationships synchronized—see Relationships and Fetched Properties in *Core Data Programming Guide*).

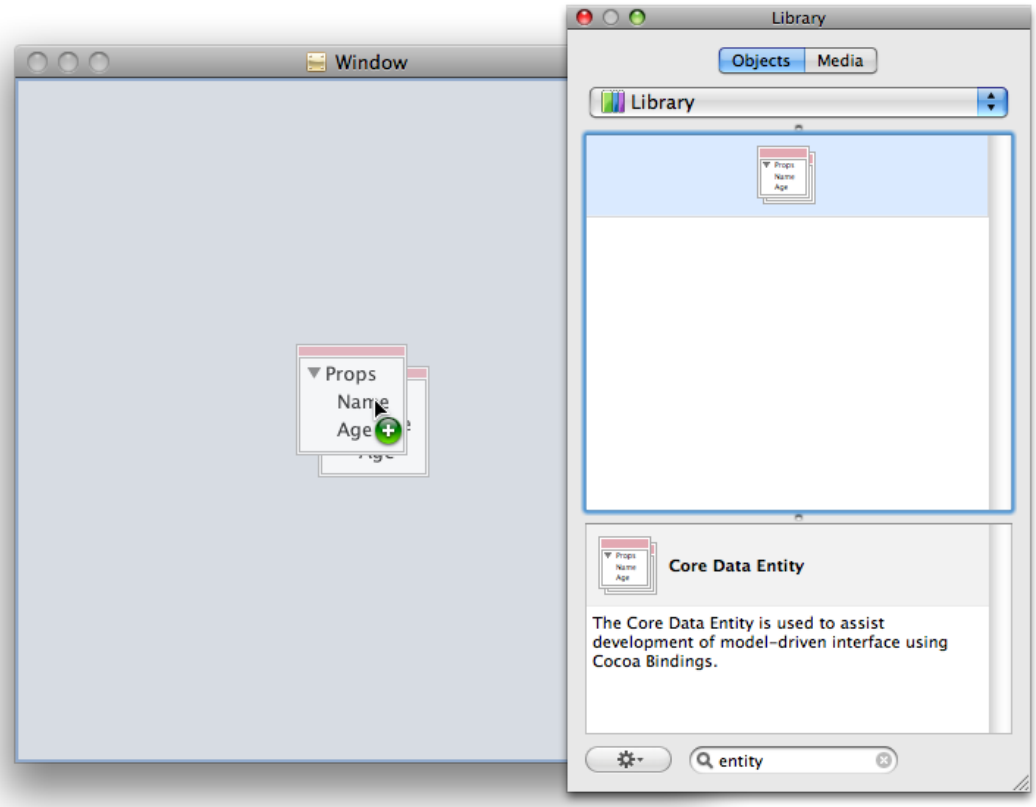
# Create the User Interface

Now that the data model is complete, you can create the user interface. In fact, you can use the model to create a user interface quickly. This provides a useful strategy for testing a model—you can create an application very quickly with little effort and use it to exercise the model.

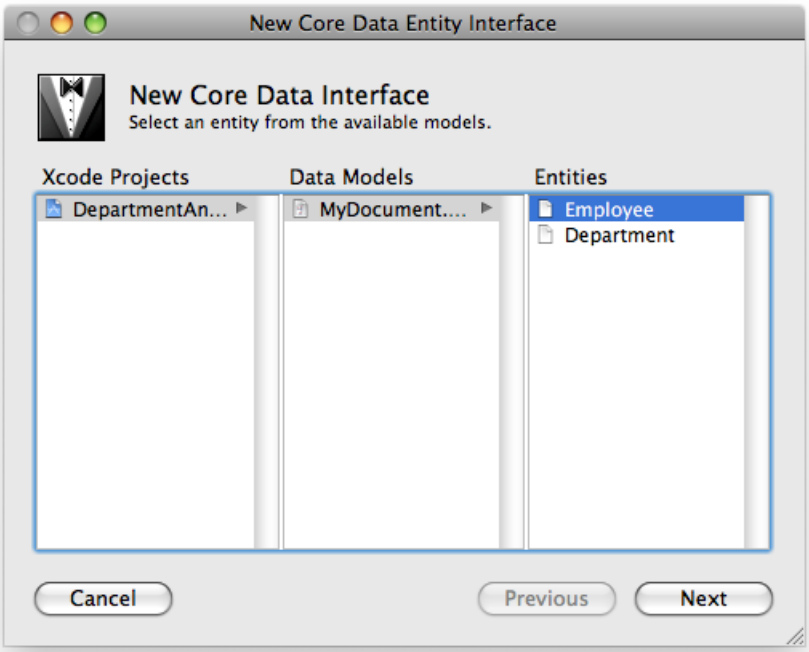
Open MyDocument.nib in Interface Builder. First remove the "Your document contents here" text field from the document window, then follow these steps:

1. Ensure that you can see the user interface window.
2. In the Library, find the Core Data Entity. Drag the entity icon over the document window, as shown in Figure 2-3 (page 18).

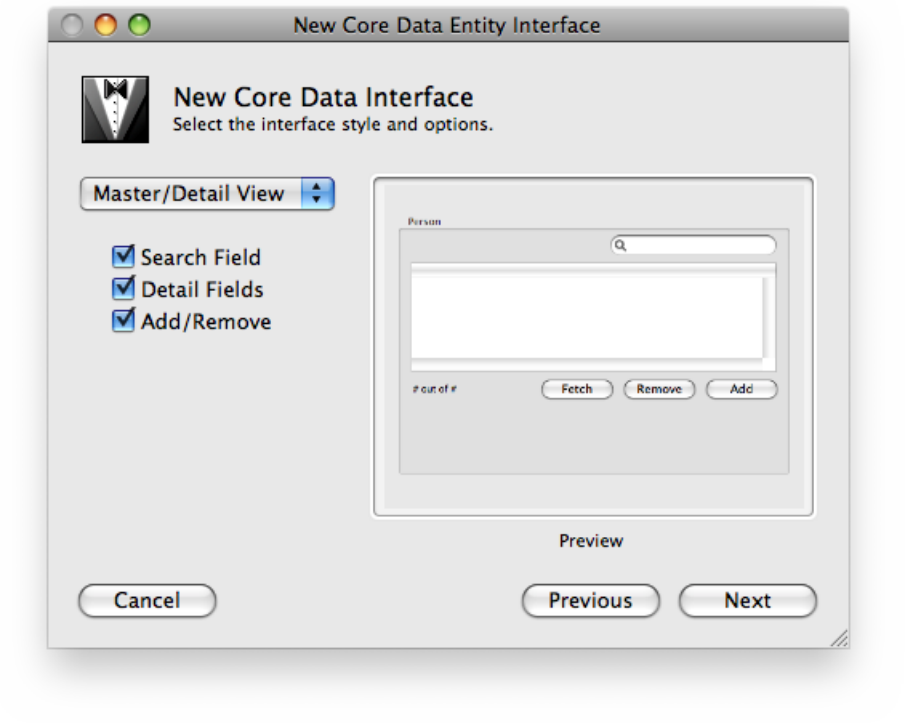
**Figure 2-3** Interface Builder with Core Data Entity



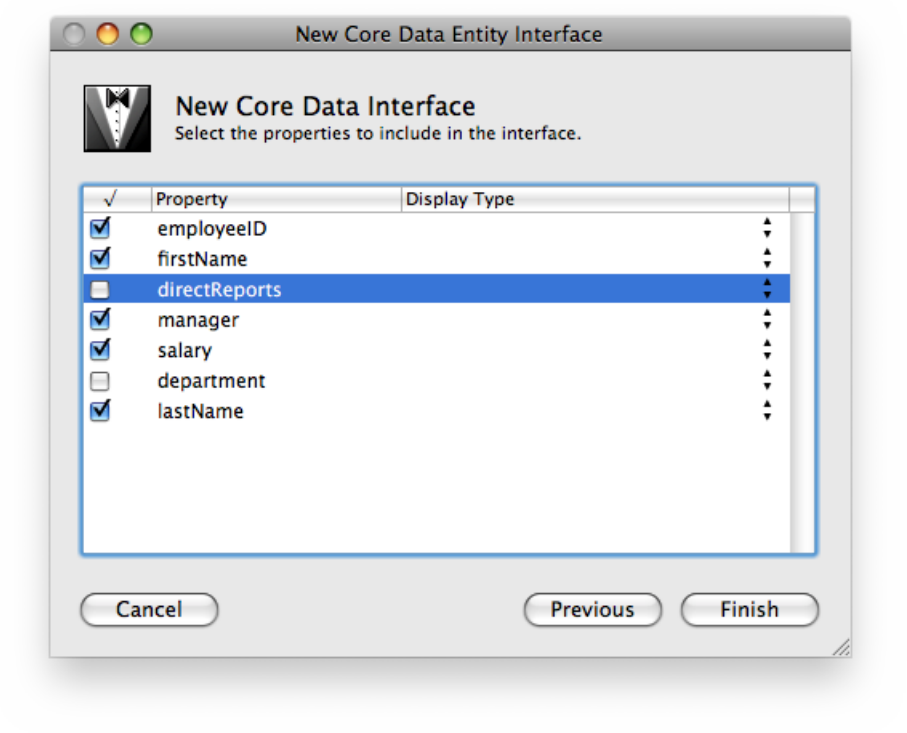
3. When you release the mouse, you are presented with the New Core Data Interface panel, which lists the currently-open Xcode projects and their associated data models. From your project, select the MyDocument model, then select the Employee entity, and finally press Next.



4. With the next panel, you choose the interface style. Select the Master/Detail view with Search field, Detail fields, and Add and Remove buttons, then press Next.

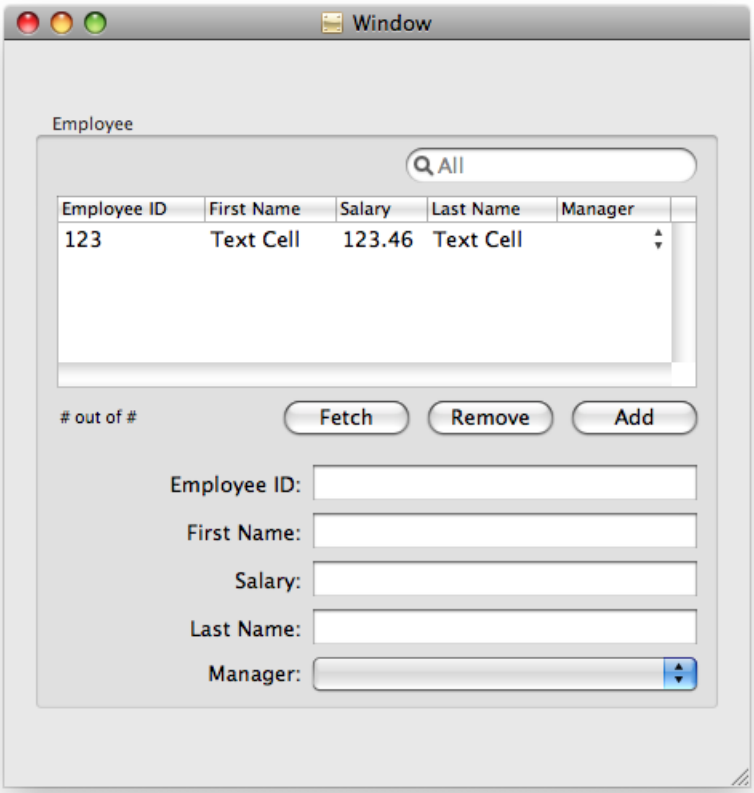


5. In the next panel, you choose which Employee properties you want to be displayed in the user interface. Choose all except `department` and `directReports`, then press Finish.



6. Interface Builder automatically creates a user interface, as illustrated in Figure 2-4. The interface contains a table view, search field, text fields for individual attributes, pop-up menu for Manager, and buttons to add, remove, and fetch employees. Interface Builder also adds array controllers to the nib file to manage the entities.

**Figure 2-4** Default automatic user interface




7. Interface Builder actually does more than is required for this tutorial. Delete the Employee ID, Salary, and Manager columns in the table view (and, if you wish, the Fetch button since it won't be used). You can also rearrange the user interface if you want; for example, you could put the first name and last name text fields next to each other.

8. Save the nib file.

You should take some time to investigate both the connections in the nib file and the behavior of the application. The user interface is created almost entirely using bindings. In particular you should note:

- The array controllers' `managedObjectContext` binding is bound to the File's Owner's `managedObjectContext` (see the Bindings pane of the `NSArrayController` Inspector).
- The array controllers are set to manage an entity, not a class (see the Attributes pane of the Inspector).
- The search field has been populated with a number of predicates—one for each attribute, and an additional one to search all attributes.

- There are two array controllers to manage collections of Employees—one for the table view, and one for the pop-up menu.

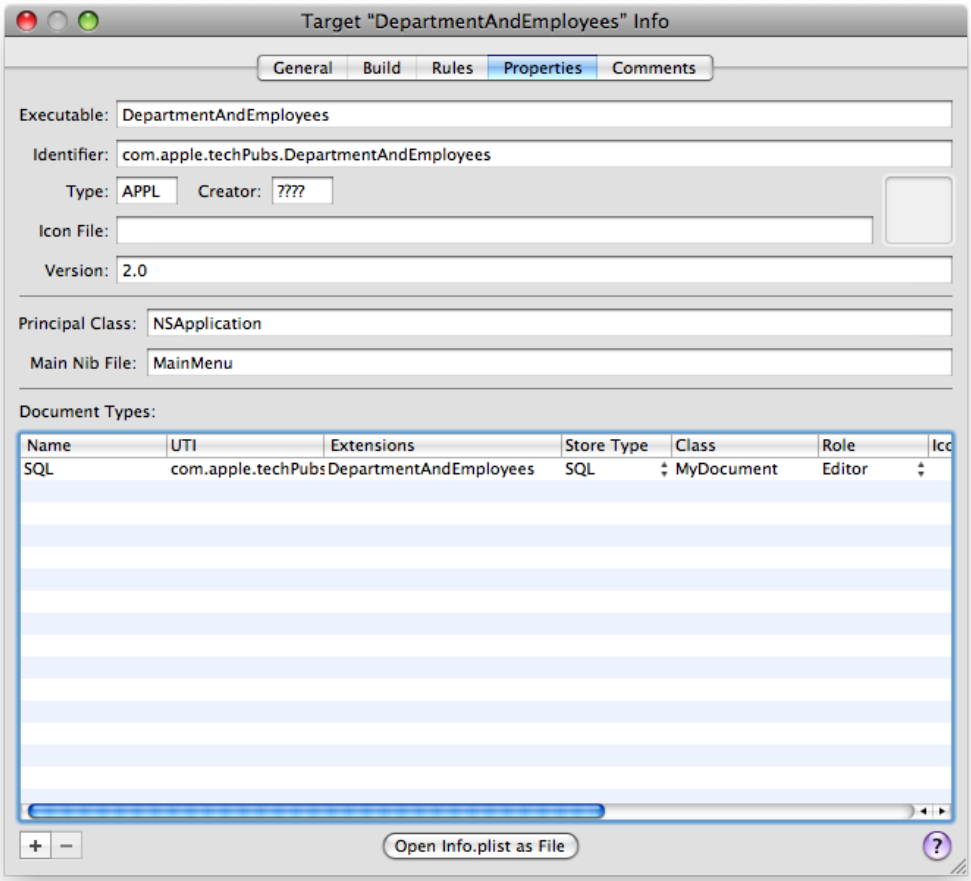
## Set the File Extension and Type

Core Data supports several types of data store—XML, SQLite, and binary, as well as your own custom store type. Each has their own advantages and disadvantages. XML, for example, is (to an extent) human-readable, which may be particularly useful during the early stages of development when you want to ensure that data you expected to be saved to a file has in fact been stored. The XML store, however, is verbose and comparatively slow. The SQLite store is smaller and more efficient. In particular, it has the unique advantage that its entire contents do not have to be read into memory when it is opened.

By default, the project is configured with three document types, one for each native Core Data persistent store type. You change these in Xcode using the Properties pane in the Target Info window as illustrated in Figure 2-5. You set the extension as you would for any other document in a Cocoa document-based application and select the desired store type from the pop-up menu in the Store Type menu. Typically SQLite represents the best choice for a deployed application.

Spotlight metadata importers are associated with document types by specifying the uniform type identifiers (UTIs—see *Uniform Type Identifiers Overview*) that they can extract data from (see Extracting Metadata from Documents). If you plan to write a custom importer you should therefore specify a suitable UTI for your document type.

**Figure 2-5** Properties pane of the Target Info window




You should also then add a UTI exported type declaration to the `Info.plist` file. You can use Xcode's plist editor, or open the Info.plist file as a plain text file and add XML elements as described below:

Using the plist editor:

- Exported Type UTIs (1 item)
  - Conforms to UTIs
    - public.data
  - Description

    SQLite file format for DepartmentAndEmployees application
  - Identifier

    com.apple.techPubs.DepartmentAndEmployees
  - Equivalent Types (1 item)
    - public.filename-extension

DepartmentAndEmployees

Editing the XML as plain text:

```
<key>UTExportedTypeDeclarations</key>
<array>
    <dict>
        <key>UTTypeConformsTo</key>
        <array>
                <string>public.data</string>
        </array>
        <key>UTTypeDescription</key>
       <string>SQLite file format for DepartmentAndEmployees application</string>
        <key>UTTypeIdentifier</key>
        <string>com.apple.devpubs.DepartmentAndEmployees</string>
        <key>UTTypeTagSpecification</key>
        <dict>
            <key>public.filename-extension</key>
            <array>
                <string>DepartmentAndEmployees</string>
            </array>
        </dict>
    </dict>
</array>
```

Finally, since this application will run only on Mac OS X v10.5 and later, add a value for the minimum system version.

Using the plist editor:

- Minimum system version

  10.5

Editing the XML as plain text:

```
<key>LSMinimumSystemVersion</key>
<string>10.5</string>
```

# Build and Test

You can now build and test your application. You should find that it is (almost) fully functional! You can add and remove employees, and edit their attributes. You can set an employee's manager using the pop-up menu. The application supports undo and redo, the search field works, and you can save and open documents. If you make a mistake and enter, for example, a negative salary, then try to save the document, the application presents an alert.

And you haven't written any code yet. . .

As you test the application, however, you may start to notice some problems. The popup menu displays only the employees' first names. First, this may not be the best way of identifying a particular employee. Second, if more than one employee share the same name, it is impossible to distinguish between them.

There are several steps that are necessary to fix these problems, all of which require a custom class for the Employee entity—which in turn requires a change to the managed object model. Specifically, you need to do the following (the implementation of these steps is described in the next section).

1. In a custom class for the Employee entity, define a custom method to return a unique derived property—a string that contains the employee's full name and employee ID.
2. Implement the relevant key-value observing method to ensure that the derived property is updated when any of its components is changed.
3. Implement a method to set a new employee ID when a new employee is created.

# What Happened?

The title "Creating the Project, Model, and Interface" (page 15) is somewhat misleading. More than just "creating the project," in a very few steps you have completed the main task goals. In the next chapters you will refine the implementation and provide additional user-friendly behavior. Now, though, it is worth briefly reviewing what has happened.

The example so far illustrates the focus of Core Data. Your primary task has been to specify the data model in your application. The Core Data framework (in conjunction with Cocoa bindings for the user interface) has provided the infrastructure to support most of the other required behaviors.

Most of the Core Data infrastructure is hidden by the persistent document. `NSPersistentDocument` implements a number of methods to support integration with Core Data. From the perspective of the typical developer, the most important method is `managedObjectContext`. The managed object context serves as the gateway to the rest of the Core Data infrastructure. It also provides the document's undo manager. `NSPersistentDocument` takes care of creating and setting the type of the persistent store, saving and loading data, and configuring the Core Data stack. For many applications you do not need to customize its behavior further. Later in this tutorial, however, you will override some of `NSPersistentDocument`'s methods to customize the store.

# Creating a Custom Employee Class

There are three parts to implementing and using a custom Employee class for the application. First, you create the class files and update the managed object model. Second, you implement the accessor method for the derived value, and ensure that key-value observing notifications are sent when any of the value's components changes. Third, you implement a method to initialize the employee ID for new employees.

## The Employee Class

1. In Xcode, create a custom managed object class using the data modeling tool by performing the following steps:

    a. In Xcode, select the data model and ensure that it is the frontmost editor—for example, simply click inside the model diagram view. (Xcode does not display the Managed Object Class option in the next step unless you do this.)

    b. Select File > New File to show the New File Assistant. In the file type outline view, select Cocoa > Managed Object Class and press Next.

    c. In the subsequent pane select the current project and target, then again press Next.

    d. In the subsequent pane (Managed Object Class Generation), select the Employee entity. Make sure the Generate Accessors and Generate Obj-C 2.0 Properties checkboxes are selected, and that the Generate validation methods checkbox is not selected.

    e. Press Finish. Xcode creates the files for the Employee class.

    Take a moment to look at the files. The attributes and relationships themselves are declared as Objective-C properties and the implementations are specified as dynamic. Similarly, the to-many relationship accessors are declared in a category, but there's no implementation. Core Data dynamically generates the implementations for you at runtime. The declarations provided in the header file make sure you don't get compiler warnings when you invoke the methods. For more details, see Managed Object Accessor Methods in *Core Data Programming Guide*.

2. In the data model (use the entity detail pane, or edit the name directly in the Class column in the entity browser), change the class name for the Employee entity from `NSManagedObject` to `Employee`.

## Support for the Derived Value

The value of `fullNameAndID` is a concatenation of, and hence dependent on, the values of `lastName`, `firstName`, and `employeeID`. To ensure that the derived value is updated whenever any of its components changes, you must implement a key-value observing method that specifies that a key is dependent on others:

you can override `keyPathsForValuesAffectingValueForKey:`, or more simply (given that in this case there is only a single property that is dependent on other properties) implement `keyPathsForValuesAffecting<Key>` (see the documentation for `keyPathsForValuesAffectingValueForKey:` for details).

## Steps

1. In the Employee class's header file, add a declaration for property:

   ```
   @property (nonatomic, readonly) NSString *fullNameAndID;
   ```

   Note that this is the only modification that you need to make to the header file. In particular, there is no need to add any instance variables.

2. In the implementation file, implement the `fullNameAndID` method. It concatenates the first and last names and the employee ID, as illustrated in the example below.

   ```
   - (NSString *)fullNameAndID {
       return [NSString stringWithFormat:@"%@, %@ (%@)",
               self.lastName, self.firstName, self.employeeID];
   }
   ```

   Notice how you can invoke the custom accessors directly (here using the dot syntax) even though you haven't provided an implementation.

3. In the Employee class, implement `keyPathsForValuesAffectingFullNameAndID` as follows:

   ```
   + (NSSet *)keyPathsForValuesAffectingFullNameAndID {
       return [NSSet setWithObjects:
               @"lastName", @"firstName", @"employeeID", nil];
   }
   ```

4. In the nib file, change the `contentValues` binding for the pop-up menu. Set the model key path to `fullNameAndID` (the other values remain the same).

You may also add `fullNameAndID` to the data model as a transient string attribute. (In this case—since the value is solely read-only and dependent on other attributes—there's no functional benefit, but it is worth doing so that the model more fully communicates the entity's behavior.)

## Build and Test

Build and run the application again. You should find that the manager pop-up properly displays the full name and ID of each employee, and that the menu item titles update as you change any of the individual components.

What the steps so far have not addressed, however, is the need to ensure that the value of fullNameAndID is unique when new employees are added.

# Initializing the Employee ID

The application could benefit from a means of initializing a managed object when it is created—or more specifically, the first time it is added to the object graph. You shouldn't use the class's designated initializer—`initWithEntity:insertIntoManagedObjectContext:`—for this since it's called each time the object is instantiated (when it is first created, and whenever it is subsequently retrieved from the persistent store). Core Data provides a special initialization method, `awakeFromInsert`, which is called once and only once in the lifetime of a managed object, on the first occasion it is inserted into a managed object context. This may be useful to, for example, set the creation date of a record. Contrast this with `awakeFromFetch`, which is called on subsequent occasions an object is fetched from a data store.

## Implement awakeFromInsert

The following implementation is crude (and should not be used in a production application—the initial tempID reverts to `1` every time the application is launched), it serves, however, to quickly illustrate the principle. You use the primitive accessor method in `awakeFromInsert` to ensure that the change to the property value is not recorded as a separate undoable action.

```
- (void)awakeFromInsert {
    static NSInteger tempID = 1;

    [super awakeFromInsert];
    self.primitiveEmployeeID = [NSNumber numberWithInteger:tempID++];
}
```

You also need to declare the primitiveEmployee property in the header file, and specify the implementation as dynamic in the implementation file:

```
// Employee.h
@property (retain) NSNumber *primitiveEmployeeID;

// Employee.m
@dynamic primitiveEmployeeID;
```

## Build and Test

Build and run the application again. You should find that as new employees are added to the document, the employee ID is set, and the ID is incremented for each new employee. More importantly, it is now possible to differentiate between all the employees in the Managers pop-up menu.

# What Happened?

Most of the task goals have now been met—primarily by creating a custom class to represent the Employee entity and implementing business logic.

A subtle point here is the interaction between the model and the employees array controller. Recall that you use the array controller to add new employees to the document. When the user interface was created, however, it was not configured to manage instances of a particular class. Instead it was configured to manage an entity (in this case, Employee). When you first built and tested the application, the model specified that employees should be represented by `NSManagedObject`. When the array controller created a new employee, therefore, it created a new instance of `NSManagedObject` (and set its entity description accordingly). After you updated the model, however, to specify that employees be represented by Employee, when the array controller created a new employee, it created a new instance of Employee. You will see in principle how this works in the next section when you create an instance of the Department entity.

# Code Listing for the Employee Class

The complete listing for the implementation of Employee class up to this point is given in Listing 3-1.

**Listing 3-1** Implementation of the Employee class

```
#import <Cocoa/Cocoa.h>


@interface Employee : NSManagedObject {

}

@property (retain) NSNumber *employeeID;
@property (retain) NSNumber *primitiveEmployeeID;
@property (retain) NSString *firstName;
@property (retain) NSString *lastName;
@property (retain) NSNumber *salary;

@property (retain) NSManagedObject *department;
@property (retain) NSSet *directReports;
@property (retain) Employee *manager;

@property (nonatomic, readonly) NSString *fullNameAndID;


@end


@interface Employee (CoreDataGeneratedAccessors)
- (void)addDirectReportsObject:(Employee *)value;
- (void)removeDirectReportsObject:(Employee *)value;
- (void)addDirectReports:(NSSet *)value;
- (void)removeDirectReports:(NSSet *)value;
@end



@implementation Employee

@dynamic employeeID;
@dynamic primitiveEmployeeID;
@dynamic firstName;
@dynamic lastName;
```

```
@dynamic salary;
@dynamic department;
@dynamic directReports;
@dynamic manager;

+ (NSSet *)keyPathsForValuesAffectingFullNameAndID {
    return [NSSet setWithObjects:
            @"lastName", @"firstName", @"employeeID", nil];
}

- (void)awakeFromInsert {
    static int tempID = 1;
    self.primitiveEmployeeID = [NSNumber numberWithInteger:tempID++];
}


- (NSString *)fullNameAndID {
    return [NSString stringWithFormat:@"%@, %@ (%@)",
            self.lastName, self.firstName, self.employeeID];
}

@end
```

# Optional Extra—Sorting the Managers Popup

The content of the pop-up button that displays the list of employees is currently unsorted. This can make it difficult to find an employee to set as another's manager. You can ensure that the pop-up menu's content is sorted by creating a sort descriptor to associate with the array controller that manages the collection of managers and rearranging the controller's contents prior to displaying the pop-up.

1. Add instance variables and outlet properties to the MyDocument class header file for the pop-up button and the managers array controller.

```
NSArrayController *managersArrayController;
NSPopUpButton *managerPopup;
NSArrayController *employeeTableController;

@property (nonatomic, retain) IBOutlet NSArrayController *managersArrayController;
@property (nonatomic, retain) IBOutlet NSPopUpButton *managerPopup;
@property (nonatomic, retain) IBOutlet NSArrayController *employeeTableController;
```

2. In the MyDocument class implementation file, synthesize the new properties and release them in the `dealloc` method.

3. In Interface Builder, make the connections as appropriate—connect the File's Owner's new `managerPopup` outlet to the pop-up menu, and the `managersArrayController` outlet to the Employees array controller that provides the content for the pop-up menu.

4. In Interface Builder, select the Auto Rearrange Content check box in the attributes inspector for the managers array controller.

5. In the MyDocument class, implement a `windowControllerDidLoadNib:` method that sets an array of sort descriptors (actually an array containing a single sort descriptor) for the managers array controller:

```
- (void)windowControllerDidLoadNib:(NSWindowController *)windowController {
    [super windowControllerDidLoadNib:windowController];

    // Create a sort descriptor to sort on "fullNameAndID"
    NSSortDescriptor *sortDescriptor = [[NSSortDescriptor alloc]
            initWithKey:@"fullNameAndID" ascending:YES];

    // Set the sortDescriptors for the managers array controller
   [managersArrayController setSortDescriptors:[NSArray arrayWithObject:sortDescriptor]];
}
```

Now build and test the application. You should find that as you add and edit employees, whenever you activate the managers pop-up button its contents are sorted alphabetically.

# Adding a Department Object

The original task specification stated that each document represents an individual department and the employees associated with it. Thus far, however, the only actions that have been taken with respect to departments has been to remove all references to them. In this section you add a Department object to the document—ensuring that only one department is associated with the document—and reconfigure the user interface appropriately.

## Creating the Department

When a new document is created, you need to create a Department object, avoiding undo registration (so that a new document does not appear edited when it is first presented). If the user opens a saved document, the Department object should already exist (it is retrieved from the persistent store). `NSDocument` provides a method—`initWithType:error:`—that is called only when a new document is created, not when it is subsequently reopened. You can therefore create the Department object in this method, and be assured that when a document is reopened a new Department object will not be created.

### Steps

1. In the MyDocument class header file, add an instance variable, `department`, of type `NSManagedObject`, and a corresponding property. The class declaration should now look like this:

```
@interface MyDocument : NSPersistentDocument
{
    NSManagedObject *department;
}
@property (nonatomic, assign) NSManagedObject *department;
@end
```

2. In the MyDocument class, synthesize the `department` property.

```
@synthesize department;
```

3. In the MyDocument class implementation file, add the instance method `-(id)initWithType:error:`. The first step is to set `self` to the result of calling the superclass's implementation, then check to ensure that `self` is not `nil`. The remainder of the implementation described in the following steps is contained within the conditional.

```
- (id)initWithType:(NSString *)type error:(NSError **)error {
    self = [super initWithType:type error:error];
    if (self != nil) {
        // implementation continues...
    }
    return self;
}
```

4. To create a new instance of department, it is easiest to use the `NSEntityDescription` convenience method `insertNewObjectForEntityForName:inManagedObjectContext:`. The method requires as its second argument a managed object context. You get this from the document itself. The method returns the new object.

```
NSManagedObjectContext *managedObjectContext = [self managedObjectContext];
self.department = [NSEntityDescription
insertNewObjectForEntityForName:@"Department"
                inManagedObjectContext:managedObjectContext];
```

   Note that this illustrates the strategy the employees array controller takes to create a new object. You don't specify the class of the new object, you specify its entity, just as you specify an entity for the array controller.

5. When you insert the new object into the managed object context it registers the event with its undo manager, so when a new document is created it would appear dirty (edited). To prevent this (and to stop the user from undoing the creation and insertion of the department) you disable undo registration before inserting the new managed object then re-enable it afterwards. Invoke `processPendingChanges` on the managed object context to ensure changes are propagated.

   After the line `NSManagedObjectContext *managedObjectContext = ...` disable undo registration:

```
[[managedObjectContext undoManager] disableUndoRegistration];
```

   After the line `self.department = ...`, process changes and re-enable undo registration:

```
[managedObjectContext processPendingChanges];
[[managedObjectContext undoManager] enableUndoRegistration];
```

## Complete Code Listing

The complete listing for `initWithType:error:` is shown in Listing 4-1.

**Listing 4-1** The complete listing for `initWithType:error:`

```
- (id)initWithType:(NSString *)type error:(NSError **)error {
    self = [super initWithType:type error:error];
    if (self != nil) {
        NSManagedObjectContext *managedObjectContext = [self
managedObjectContext];
        [[managedObjectContext undoManager] disableUndoRegistration];
        self.department = [NSEntityDescription
insertNewObjectForEntityForName:@"Department"
                inManagedObjectContext:managedObjectContext];
        [managedObjectContext processPendingChanges];
        [[managedObjectContext undoManager] enableUndoRegistration];
    }
    return self;
}
```

# Fetching the Department

**Note:** This example follows the traditional Cocoa pattern of adding the Department as an instance variable to the document class. Using Core Data, there may be no need to do this—see "Adopting the Mediator Pattern" (page 39). The traditional pattern is used here to illustrate aspects of fetching an object. There are also other situations in which this pattern remains valid and useful.

If you need to access the department from within any of your document's methods, you need to fetch it from the persistent store.

In order to perform a fetch, you need a fetch request and a managed object context. The fetch request specifies what instances of a particular entity it is that you fetch. By implication, therefore, you also need at least an entity description. The managed object context is the gateway to the underlying persistent store coordinator and hence persistent stores.

You can define an accessor method for the department. The first thing it should do is check whether or not the department has already been fetched. If it has, return it immediately. If it has not already been fetched, create a fetch request for the Department entity and fetch from the document's managed object context.

## Steps

1. In the MyDocument class implementation file, add the instance method `-(NSManagedObject *)department`. The first step is to check whether `department` is not `nil`. If it is not, return it.

```
- (NSManagedObject *)department {
    if (department != nil) {
        return department;
    }
    // implementation continues...
    return department;
}
```

2. To use a fetch request, you need a managed object context, an `NSError` variable to pass as an argument to the fetch method, and an array variable to which the returned value is assigned. Given these, you can create the fetch request.

```
NSManagedObjectContext *moc = [self managedObjectContext];
NSError *fetchError = nil;
NSArray *fetchResults;
NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
```

3. As a minimum for the fetch request, you must specify the entity description for the entity that is to be fetched. You may also provide a predicate and an array of sort descriptors. In this case there is (or should be!) only one department to fetch, so neither a predicate nor sort orderings are required.

   You get the entity description using a convenience method—`entityForName:inManagedObjectContext:`—of `NSEntityDescription`. It takes as its arguments the name of an entity and a managed object context. It uses the context to find the persistent store coordinator, and from the model associated with the coordinator, the entity description with the specified name.

You set the entity for the fetch request, then use the context to execute the fetch.

```
NSEntityDescription *entity = [NSEntityDescription entityForName:@"Department"
            inManagedObjectContext:moc];
[fetchRequest setEntity:entity];
fetchResults = [moc executeFetchRequest:fetchRequest error:&fetchError];
```

4. If there is one object in the returned array, and there is no fetch error, the object in the array is the Department object. If these conditions are not satisfied, then something has gone wrong. If there is an error, you can display it most easily using `NSDocument`'s `presentError:` method. This, however, creates an application-modal window, so ideally you should use `presentError:modalForWindow:delegate:didPresentSelector:contextInfo:` to present a panel modal just for the window, but showing how to do that lies outside the scope of this example (it requires significant additional code and explanation that is not directly related to understanding Core Data and `NSPersistentDocument`). If there is no error, but either the result is `nil` or the count of the results array is not `1`, then something has gone wrong and the user should be alerted. Again how to do this is not shown here.

```
if ((fetchResults != nil) && ([fetchResults count] == 1) && (fetchError == nil))
 {
    self.department = [fetchResults objectAtIndex:0];
    return department;
}
if (fetchError != nil) {
    [self presentError:fetchError];
}
else {
    // should present custom error message...
}
return nil;
```

## Complete Code Listing

The complete listing for the `department` method is given in Listing 4-2.

**Listing 4-2** The complete listing for the `department` method

```
- (NSManagedObject *)department
{
    if (department != nil) {
        return department;
    }

    NSManagedObjectContext *moc = [self managedObjectContext];
    NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
    NSError *fetchError = nil;
    NSArray *fetchResults;

   NSEntityDescription *entity = [NSEntityDescription entityForName:@"Department"
                                              inManagedObjectContext:moc];

    [fetchRequest setEntity:entity];
    fetchResults = [moc executeFetchRequest:fetchRequest error:&fetchError];
```

```
    if ((fetchResults != nil) && ([fetchResults count] == 1) && (fetchError ==
 nil)) {
        self.department = [fetchResults objectAtIndex:0];
        return department;
    }

    if (fetchError != nil) {
        [self presentError:fetchError];
    }
    else {
        // should present custom error message...
    }
    return nil;
}
```

# Custom Department Methods

In this tutorial, you do not create a custom Department class. There is no custom logic and there's no need to implement custom accessor methods—Core Data generates efficient accessors for you at runtime (see Managed Object Accessor Methods). Later in the tutorial ("Setting Metadata for a Store" (page 55)), however, you need to access the document's department's name. You can do this using key-value coding:

```
NSString *departmentName = [[self department] valueForKey:@"name"];
```

but it is more efficient to use accessor methods (this example shows using the dot syntax):

```
NSString *departmentName = self.department.name;
```

(The efficiency gain in the situation described in "Setting Metadata for a Store" (page 55) is minimal, however the principle here is the important thing.) To avoid compiler warnings, though, you need to declare the accessor methods. You can do this using properties in a category of `NSManagedObject`.

- Add the following code before the `@implementation` block of `MyDocument`:

```
@interface NSManagedObject (DepartmentAccessors)
@property (retain) NSDecimalNumber *budget;
@property (retain) NSString *name;
@end
```

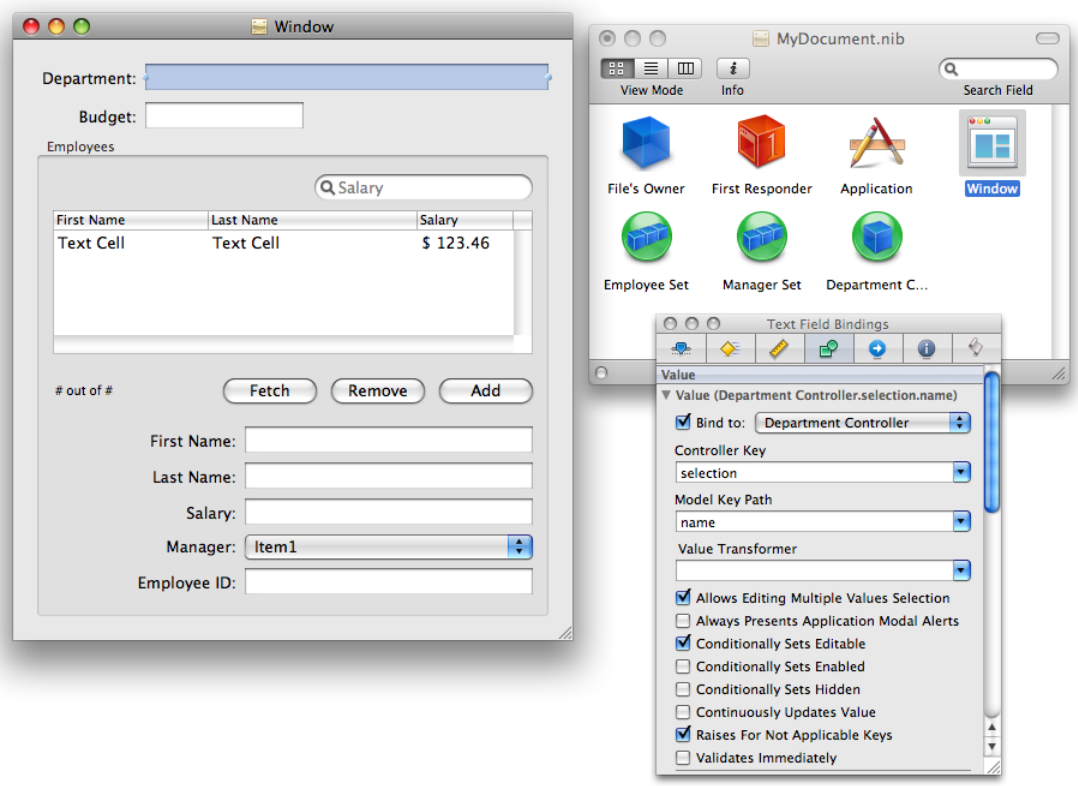
# Update the User Interface

You can now update the user interface to include the Department object. Figure 4-1 provides an example of how the user interface might look when you have finished.

1. Open the MyDocument nib file in Interface Builder. Add an `NSObjectController` instance. Its entity is Department. Bind its `managedObjectContext` to File's Owner's `managedObjectContext` and its `contentObject` to File's Owner's `department`.

2. Add two text fields to the window. Bind the value of one to the department's name (bind to the Department object controller's `selection.name`), the other to the department's budget—the latter requires a number formatter (so that the input string is converted into a number object). (To set a formatter in Interface Builder, drag a number formatter from the Cocoa library onto the text field.)

**Figure 4-1** User interface with department




You can now specify that the employee array controllers retrieve their employees not directly from the managed object context, but from the department's employees relationship. This is important to ensure that when a new employee is added, it is properly added to the department's employees relationship, and the employee's department relationship is set.

Remember to bind the array controller's `contentSet`, not its `contentArray`. Managed objects represent to-many relationships using a set, not an array. Remember also that by default, removing an object from an array controller whose content set is bound to a relationship simply removes the object from the relationship, not from the object graph. If (as is the case) you want the remove operation to act as a delete, you must enable the Deletes Objects On Remove option for the `contentSet` binding.

1. For each array controller, bind the `contentSet` to the department controller's `selection.employees`.

2. Inspect the bindings for the Employees array controller that manages the content of the table view. For the `contentSet` binding, enable the Deletes Objects On Remove option.

## Build and Test

Build and run the application again. You should find that if you set the department name and then save the document, when you reopen the document, its department's name is properly reconstituted.

# Supporting Document Revert

Since the document maintains a strong reference to the department object, you should make sure that it is let go in the case of the document being reverted. You need to implement a `revertToContentsOfURL:ofType:error:` method, as follows:

```
- (BOOL)revertToContentsOfURL:(NSURL *)inAbsoluteURL ofType:(NSString *)inTypeName
 error:(NSError **)outError {
    BOOL result = [super revertToContentsOfURL:inAbsoluteURL ofType:inTypeName
 error:outError];
    if (result) {
        self.department = nil;
    }
    return result;
}
```

# Adopting the Mediator Pattern

In the Cocoa document architecture, an `NSDocument` instance serves primarily as a model controller and one or more instances of `NSWindowController` serve as view controllers (see The Roles of Key Objects in Document-Based Applications). The mediator pattern extends this concept of distributed control—mediating controllers mediate the flow of data between view objects and model objects in an application (for a general discussion, see Cocoa Design Patterns).

In the current application, the Department instance serves as a "root" object for the graph of model objects. This is a common pattern in traditional Cocoa programming, and you would typically access other model objects via relationships from this root object through the document instance. Many developers may be more used to and more comfortable with the idea of keeping a reference to a root model object or collection. If you are using Core Data, however, the recommendation is not to do that (unless it is necessary or useful), but instead to hand responsibility for model object graph to the managed object context. If you need a reference to a particular instance, you either execute a fetch request, or (more commonly) retrieve it from the relevant object controller.

In this example, there is actually no *need* to keep an explicit reference to the department in the document instance, although it is convenient for an implementation of the paste method (see "Paste" (page 44)) and—to reiterate—here it served the useful purpose of illustrating how to fetch objects from the document's managed object context. Since there is only ever one Department record, however, the `NSObjectController` object can simply fetch it directly. Thus, instead of binding the department object controller's `contentObject` to the File's Owner, you bind only its `managedObjectContext` and configure it to automatically prepare content (see `setAutomaticallyPreparesContent:`)—this means that at runtime, the controller automatically executes a fetch to fill its content. Since there is one and only one Department instance, the correct Department

instance is retrieved. If you do need a reference to the instance, you can either fetch it or retrieve it from the object controller. You then dispense with the `department` and `setDepartment:` methods, and the explicit support for the revert method described in "Supporting Document Revert" (page 39).

## What Happened?

More of the task goals have now been met through adding an instance of Department to the document.

# Copy and Paste

Most applications support copy and paste. Copy and paste of managed objects is broadly similar to that of other objects, except that you need to be careful about how much of the object graph you copy.

The focus of this chapter is on how to copy managed objects, not how to provide an architecture for copy and paste. In this example, therefore, a simplistic approach is taken—the document object implements `cut:`, `copy:`, and `paste:`, and only supports copying of employees selected in the table view. There are many variants that could be implemented, this example illustrates just one approach. Moreover, basic Cocoa techniques such as archiving, key-value coding, and creating and setting outlets in a nib file, are not explained in detail.

## Custom Employee Logic

Although it is not strictly necessary to modify the Employee class to support copy and paste, it makes sense to implement custom business logic in one place rather than distributing it throughout the application. The main decision you must make, however, is what it means to copy an employee—what properties of an employee are copied. It seems obvious that first name, last name, and salary should be copied. In this example, it is unlikely that department, manager, or direct reports should be copied. The department is set during the paste operation. If you copy the manager or the direct reports, you are likely to end up copying the whole object graph as you add those objects to the copy. Moreover, copying related objects presents difficulties in cases where a given object may be referred to more than once—you need to ensure uniqueness in the copied graph. Finally, the employee ID requires a judgement call. Whether or not you choose to copy it depends on the semantics of copy (or particularly of paste) in the application.

To abstract out some of this logic, declare and implement a class method—`keysToBeCopied`—that returns an array of keys of attributes to be copied. To support copy of an employee both as an object and as a string (to paste into another application), declare and implement instance methods to return a dictionary and a string representation of the object.

1. In the Employee class, declare and implement a class method, `keysToBeCopied`, as follows:

```
+ (NSArray *)keysToBeCopied {
    static NSArray *keysToBeCopied = nil;
    if (keysToBeCopied == nil) {
        keysToBeCopied = [[NSArray alloc] initWithObjects:
            @"firstName", @"lastName", @"salary", @"employeeID", nil];
    }
    return keysToBeCopied;
}
```

2. In the Employee class, declare and implement an instance method, `dictionaryRepresentation`, as follows:

```
- (NSDictionary *)dictionaryRepresentation {
    return [self dictionaryWithValuesForKeys:[[self class] keysToBeCopied]];
}
```

3. In the Employee class, declare and implement an instance method, `stringDescription`, as follows. (Note that you are discouraged from overriding the `description` method.)

```
- (NSString *)stringDescription {
    NSString *stringDescription = self.fullNameAndID;
    NSString *managerString = @"none";
    Employee *manager = self.manager;
    if (manager != nil) {
        managerString = manager.fullNameAndID;
    }
    stringDescription = [stringDescription stringByAppendingFormat:
        @"; Manager: %@", managerString];
    return stringDescription;
}
```

# Copy

In order to copy, you need to know what the current selection is. You can get this information most easily from the employees array controller. You can also define a label for the employee pasteboard type.

## Steps

1. In the MyDocument class header file, declare an IBOutlet `employeeTableController` of type `NSArrayController`.

```
IBOutlet NSArrayController *employeeTableController;
```

2. In Interface Builder, import the header into the document nib file, and connect the File's Owner's `employeeTableController` outlet to the employee array controller.

3. In the MyDocument class implementation file, declare the global string `EmployeesPBoardType`. Also import the Employee header file.

```
#import "Employee.h"
NSString *EmployeesPBoardType = @"EmployeesPBoardType";
```

4. In the MyDocument class implementation file, implement a `copy:` method. It first retrieves the employee array controller's selected objects. If no objects are selected (the count of the array is 0), it returns immediately. (The count is used later if it is not 0.)

```
- (void)copy:sender {
    NSArray *selectedObjects = [employeeTableController selectedObjects];
    NSUInteger count = [selectedObjects count];
    if (count == 0) {
        return;
    }
    // implementation continues....
}
```

5. Create two mutable arrays, one to contain the dictionary representations of the objects to copy, the other to contain the string representations. Iterate over the array of selected employees, adding the appropriate representation of each object to the corresponding array.

```
NSMutableArray *copyObjectsArray = [NSMutableArray arrayWithCapacity:count];
NSMutableArray *copyStringsArray = [NSMutableArray arrayWithCapacity:count];

for (Employee *employee in selectedObjects) {
    [copyObjectsArray addObject:[employee dictionaryRepresentation]];
    [copyStringsArray addObject:[employee stringDescription]];
}
```

6. Declare the types to be copied for the general pasteboard, and set the corresponding values. Since the dictionary representation of an employee contains only property list types, you can simply create an archive of the array to set as the data for the custom pasteboard. For the string representation, concatenate the individual strings, separating them with a newline character.

```
NSPasteboard *generalPasteboard = [NSPasteboard generalPasteboard];
[generalPasteboard declareTypes:
        [NSArray arrayWithObjects:EmployeesPBoardType, NSStringPboardType, nil]
                           owner:self];
NSData *copyData = [NSKeyedArchiver archivedDataWithRootObject:copyObjectsArray];
[generalPasteboard setData:copyData forType:EmployeesPBoardType];
[generalPasteboard setString:[copyStringsArray componentsJoinedByString:@"\n"]
                     forType:NSStringPboardType];
```

## Complete Code Listing

The complete listing for the `copy:` method is shown in Listing 5-1.

**Listing 5-1** Complete listing of the `copy:` method

```
- (void)copy:sender {
    NSArray *selectedObjects = [employeeTableController selectedObjects];
    NSUInteger count = [selectedObjects count];
    if (count == 0) {
        return;
    }

    NSMutableArray *copyObjectsArray = [NSMutableArray arrayWithCapacity:count];
    NSMutableArray *copyStringsArray = [NSMutableArray arrayWithCapacity:count];

    for (Employee *employee in selectedObjects) {
        [copyObjectsArray addObject:[employee dictionaryRepresentation]];
        [copyStringsArray addObject:[employee stringDescription]];
    }

    NSPasteboard *generalPasteboard = [NSPasteboard generalPasteboard];
    [generalPasteboard declareTypes:
        [NSArray arrayWithObjects:EmployeesPBoardType, NSStringPboardType, nil]
                           owner:self];
    NSData *copyData = [NSKeyedArchiver
archivedDataWithRootObject:copyObjectsArray];
    [generalPasteboard setData:copyData forType:EmployeesPBoardType];
   [generalPasteboard setString:[copyStringsArray componentsJoinedByString:@"\n"]
```

```
                            forType:NSStringPboardType];
}
```

## Build and Test

Build and run the application.

Although you have not yet implemented support for paste within the application, you should be able to paste a string representation of the current selection into, for example, a TextEdit document or a Mail message.

# Paste

In order to paste, you create employee objects from the array of dictionaries on the pasteboard. You must insert these into the document's managed object context, and add them to the department's employees relationship.

## Steps

1. In the MyDocument class implementation file, implement a `paste:` method. It first retrieves the employee array data from the pasteboard (using the custom pasteboard type). If there is no data, return immediately.

```
- (void)paste:sender {
    NSPasteboard *generalPasteboard = [NSPasteboard generalPasteboard];
    NSData *data = [generalPasteboard dataForType:EmployeesPBoardType];
    if (data == nil) {
        return;
    }
    // implementation continues....
}
```

2. To create the new employees, you need to unarchive the array of dictionaries. You also need the document's managed object context and a way to add each new employee to the department object's employees relationship. You can address the latter requirement using key-value coding with the `mutableSetValueForKey:` method:

```
NSArray *employeesArray = [NSKeyedUnarchiver unarchiveObjectWithData:data];
NSManagedObjectContext *moc = [self managedObjectContext];
NSMutableSet *departmentEmployees = [self.department
mutableSetValueForKey:@"employees"];
```

3. For each item in the employees array, create a new employee object and establish the relationship between it and the department. The easiest way to create a new employee is using the `NSEntityDescription` class method `insertNewObjectForEntityForName:inManagedObjectContext:`. This returns a new instance of the class specified in the managed object model to represent the Employee entity. You can then set the attribute values of the new object from the dictionary using key-value coding.

To establish the relationship between the employee and department, you can either add the employee to the department's employees relationship or set the department for the employee directly. (Since the relationship is modeled in both directions, and the inverse relationships properly specified, the referential integrity is maintained automatically.) For the purpose of illustrating manipulation of a to-many relationship, do the former:

```
for (NSDictionary *employeeDictionary in employeesArray) {
    Employee *newEmployee;
    newEmployee = (Employee *)[NSEntityDescription
insertNewObjectForEntityForName:@"Employee"
                                    inManagedObjectContext:moc];
    [newEmployee setValuesForKeysWithDictionary:employeeDictionary];
    [departmentEmployees addObject:newEmployee];
}
```

## Complete Code Listing

The complete listing for the `paste:` method is shown in Listing 5-2.

**Listing 5-2** Complete listing of the `paste:` method

```
- (void)paste:sender {
    NSPasteboard *generalPasteboard = [NSPasteboard generalPasteboard];
    NSData *data = [generalPasteboard dataForType:EmployeesPBoardType];
    if (data == nil) {
        return;
    }

    NSManagedObjectContext *moc = [self managedObjectContext];
    NSMutableSet *departmentEmployees = [self.department
mutableSetValueForKey:@"employees"];
    NSArray *employeesArray = [NSKeyedUnarchiver unarchiveObjectWithData:data];

    for (NSDictionary *employeeDictionary in employeesArray) {
        Employee *newEmployee;
        newEmployee = (Employee *)[NSEntityDescription
insertNewObjectForEntityForName:@"Employee"
                                        inManagedObjectContext:moc];
        [newEmployee setValuesForKeysWithDictionary:employeeDictionary];
        [departmentEmployees addObject:newEmployee];
    }
}
```

## Build and Test

You should now be able to compile and test the application. You should be able to copy selected employees and paste them into either the same or a different document. You should also notice that undo and redo work appropriately.

# Cut

In order to cut, you first copy the existing selection, then delete it. To delete the selected employees, you delete them from the managed object context.

## Steps

1. In the MyDocument class implementation file, implement a `cut:` method. It first calls `copy:`.

```
- (void)cut:sender {
    [self copy:sender];

    // implementation continues....
}
```

2. To delete the employees, you need the document's managed object context. You then need to retrieve the array of selected objects from the employee table controller. For each item in the array of selected employees, delete it from the context:

```
NSArray *selectedEmployees = [employeeTableController selectedObjects];

if ([selectedEmployees count] == 0) {
    return;
}
NSManagedObjectContext *moc = [self managedObjectContext];

for (Employee *employee in selectedEmployees) {
    [moc deleteObject:employee];
}
```

Alternatively, since you have a reference to the employee controller, you could send it a `removeObject:` message for each selected employee. To use this pattern you must ensure that the Deletes Object on Remove option is set for the `contentSet` binding. (Objects are deleted automatically if the array controller's content is fetched automatically. In this case, however, the `contentSet` is bound to the department's employees relationship, so unless the Deletes Object on Remove option is set, `removeObject:` removes the object only from the relationship, not from the object graph.)

Note that, since Employee's department relationship delete rule is set to Nullify, there is no need to remove the employees from the department's employees relationship—this is performed automatically by the framework.

## Complete Code Listing

The complete listing for the `cut:` method is shown in Listing 5-3.

**Listing 5-3** Complete listing of the `cut:` method

```
- (void)cut:sender {
    [self copy:sender];
    NSArray *selectedEmployees = [employeeTableController selectedObjects];
```

```
    if ([selectedEmployees count] == 0) {
        return;
    }
    NSManagedObjectContext *moc = [self managedObjectContext];

    for (Employee *employee in selectedEmployees) {
        [moc deleteObject:employee];
    }
}
```

## Build and Test

You should now be able to compile and test the application. You should be able to cut selected employees from one document and paste them into either the same or a different document. You should also notice that undo and redo work appropriately.

# Localizing and Customizing Model Property Names and Error Messages

By default, if an error occurs the user is presented an alert indicating what the problem was. Although useful, the text in the alert may not be very user friendly. If there was a validation error, the name of the property that failed validation is given as defined in the model (for example, "firstName") rather than something more natural (such as "First name")—this may be especially unhelpful for users whose native language is not that of the developer. Moreover, if more than one error occurs, the alert simply states that many errors have occurred. In order to give the user more information and help them to fix the problems, you can customize the display of property names and multiple errors.

## Customizing and Localizing Model Names

You can customize and localize entity and property names by adding a model strings file to your project resources (see *Core Data Programming Guide* > Using a Managed Object Model). You set the name of the strings file to `<ModelName>Model.strings`. For properties where there is only one entity with the given property name, the keys in the file take the form `Property/<NonLocalizedPropertyName>`—if there is a chance of a conflict, you can use `Property/<NonLocalizedPropertyName>/Entity/<NonLocalizedEntityName>`.

### Steps

1. Select the Resources folder of your project. Create a new empty file by choosing File > New File then selecting Empty File in Project. Press Next, then name the file `MyDocumentModel.strings`.

2. Make the file localizable (in the Inspector (File Info window), click Make File Localizable). You should also ensure that the file is saved in UTF-16 encoding (see "Part III: Editing" > "Editing File Information" > "Choosing File Encodings" in the Xcode User Guide).

3. Add key-value pairs for localized representations of the Employee entity's properties of the form `"Property/NonLocalizedPropertyName" = "Pretty Property Name";` as illustrated in the following examples. Note that it is important to include the semicolon at the end of each line.

```
"Property/firstName" = "First Name";
"Property/lastName" = "Last Name";
"Property/salary" = "Salary";
```

4. Add a second localized version of the file (in the Xcode inspector, click Add Localization). Choose a localization for a language into which you can translate the property names, or use "fr" (for French) and follow this example:

```
"Property/firstName" = "Prénom";
"Property/lastName" = "Nom de famille";
"Property/salary" = "Salaire";
```

## Build and Test

Build and run the application. Add a single new employee to a document, and delete the first name. Save the document. You should see an alert that includes the text, "First Name is a required value".

Inspect the executable and add the argument `-AppleLanguages "(fr)"` (or instead of "fr" use whatever locale identifier you chose earlier). Run the application again, and add a single new employee to a document. Delete the value for the first name, and try to save. This time you should see an alert that displays the localized version of the first name key. Other parts of the interface may also be localized, depending on what locales you installed on your system, and what locale you chose for this test.

# Customizing the Document Alert Panel

Cocoa provides a sophisticated error-handling architecture. Exactly how you deal with an error, and which object receives a message in the event of an error, is described in detail in *Error Handling Programming Guide*. In some cases, for example, if a document fails to open, it may be the responsibility of the `NSDocumentController` object or of the application object to deal with the error. This section focuses on specific errors that may be dealt with by the document itself.

There may be many reasons why an error occurs, and many errors may occur simultaneously—for example, if a user enters several data values that fail validation then tries to save a document. In the event that an error occurs, Core Data provides a rich set of information that describes the problem in an `NSError` object, but if many things go wrong at the same time the actual description provided to the user may be impoverished. `NSDocument` provides a method—`willPresentError:`—that you can override to customize an error message. The method's argument is the original error. You first, therefore, need to determine whether an error is one that you want to handle in a custom manner. In the case of the `willPresentError:` method, if it's not an error you want to handle in a custom manner you then simply return it, otherwise analyze the error to determine what to do.

## Steps

1. In the My Document class, create a stub entry for the method `willPresentError:`.

```
- (NSError *)willPresentError:(NSError *)inError {
    // implementation continues...
}
```

2. To customize error messages for Core Data, you first check the error domain. If it is not a Core Data error, (in this example) simply return the original error.

```
if (![[inError domain] isEqualToString:NSCocoaErrorDomain]) {
    return inError;
}
```

3. Core Data provides errors for a range of different situations. In this example, the only errors of interest are validation errors. The error codes for Core Data validation errors lie within a range delimited by `NSValidationErrorMinimum` and `NSValidationErrorMaximum` (declared in `NSError.h`—error codes and `userInfo` dictionary keys specific to Core Data are declared in `CoreData/CoreDataErrors.h`). If the error code is not within this range, again return the original error. (This step is not strictly necessary—the only test required is the next one—but is a useful example.)

```
NSInteger errorCode = [inError code];
if ((errorCode < NSValidationErrorMinimum) ||
            (errorCode > NSValidationErrorMaximum)) {
    return inError;
}
```

4. If there are multiple validation errors, the error is an `NSValidationMultipleErrorsError`. If the error is not an `NSValidationMultipleErrorsError`, then there is only one error message to report, so again return the original error.

```
if (errorCode != NSValidationMultipleErrorsError) {
    return inError;
}
```

5. If the error is an `NSValidationMultipleErrorsError`, its `userInfo` dictionary contains an array of the original error under the key, `NSDetailedErrorsKey`.

   For this example, present error messages for no more than three validation errors at a time. You can do so simply by concatenating the localized description for each error. You could instead construct a more customized, user-friendly error by examining the error code of each individual error.

```
NSArray *detailedErrors = [[inError userInfo] objectForKey:NSDetailedErrorsKey];

unsigned numErrors = [detailedErrors count];
NSMutableString *errorString = [NSMutableString stringWithFormat:@"%u validation
 errors have occurred", numErrors];

if (numErrors > 3) {
    [errorString appendFormat:@".\nThe first 3 are:\n"];
}
else {
    [errorString appendFormat:@":\n"];
}
NSUInteger i, displayErrors = numErrors > 3 ? 3 : numErrors;
for (i = 0; i < displayErrors; i++) {
    [errorString appendFormat:@"%@\n",
            [[detailedErrors objectAtIndex:i] localizedDescription]];
}
```

6. Finally, create a new error based on the original error and the new description and return it.

```
NSMutableDictionary *newUserInfo = [NSMutableDictionary
            dictionaryWithDictionary:[inError userInfo]];
[newUserInfo setObject:errorString forKey:NSLocalizedDescriptionKey];

NSError *newError = [NSError errorWithDomain:[inError domain]
                                        code:[inError code]
                                    userInfo:newUserInfo];
return newError;
```

## Complete Code Listing

The complete `willPresentError:` method is shown in Listing 6-1.

**Listing 6-1** Complete listing of the `willPresentError:` method

```
- (NSError *)willPresentError:(NSError *)inError {

    // The error is a Core Data validation error if its domain is
    // NSCocoaErrorDomain and it is between the minimum and maximum
    // for Core Data validation error codes.

    if (!([[inError domain] isEqualToString:NSCocoaErrorDomain])) {
        return inError;
    }

    NSInteger errorCode = [inError code];
    if ((errorCode < NSValidationErrorMinimum) ||
                (errorCode > NSValidationErrorMaximum)) {
        return inError;
    }

    // If there are multiple validation errors, inError is an
    // NSValidationMultipleErrorsError. If it's not, return it

    if (errorCode != NSValidationMultipleErrorsError) {
        return inError;
    }

    // For an NSValidationMultipleErrorsError, the original errors
    // are in an array in the userInfo dictionary for key NSDetailedErrorsKey
    NSArray *detailedErrors = [[inError userInfo]
objectForKey:NSDetailedErrorsKey];

    // For this example, only present error messages for up to 3 validation
errors at a time.

    unsigned numErrors = [detailedErrors count];
    NSMutableString *errorString = [NSMutableString stringWithFormat:@"%u
validation errors have occurred", numErrors];

    if (numErrors > 3) {
        [errorString appendFormat:@".\nThe first 3 are:\n"];
    }
    else {
        [errorString appendFormat:@":\n"];
    }
    NSUInteger i, displayErrors = numErrors > 3 ? 3 : numErrors;
    for (i = 0; i < displayErrors; i++) {
        [errorString appendFormat:@"%@\n",
            [[detailedErrors objectAtIndex:i] localizedDescription]];
    }

    // Create a new error with the new userInfo
    NSMutableDictionary *newUserInfo = [NSMutableDictionary
                dictionaryWithDictionary:[inError userInfo]];
    [newUserInfo setObject:errorString forKey:NSLocalizedDescriptionKey];

    NSError *newError = [NSError errorWithDomain:[inError domain] code:[inError
 code] userInfo:newUserInfo];

    return newError;
```

```
}
```

## Build and Test

Build and run the application again. Add two employees to a document, and set some invalid values—for example, set their salaries to negative numbers. When you try to save the document, you should find that an alert is presented that lists the validation failures. Add two more employees and try the same test again to ensure that the alert displays just the first three errors.

Comment out the `willPresentError:` method and build and run the application again. Contrast the alert presented without the custom method with that presented using your custom method.

Create a new directory and remove write permissions for yourself—now try to save the document into that directory. What error is presented? What error code is generated?

# What Happened?

You have made the application more user friendly. Core Data provides rich information when something goes wrong. You can typically help the user by making it as clear as possible what the problem is, so that the user may have a chance to fix it.

You investigated Core Data's error handling, in particular you discovered that errors are given a rich description. It's possible to find out whether a given error is a Core Data error, if so whether it's a validation error, and finally, if it's a multiple validation error, what were the original errors.

# Document Metadata

Spotlight provides users with a means of searching for files quickly and easily. To support this, you need to associate metadata with your documents. Core Data makes it easy to do this and to write the necessary importer.

## Setting Metadata for a Store

Note that setting the metadata only queues up the information to be saved when the store is next saved—it is not written out immediately.

### Steps

1. You identify a store by its URL. Since there is more than one place that this code will be used, define a method in MyDocument to abstract the logic.

```
- (BOOL)setMetadataForStoreAtURL:(NSURL *)url {
    // implementation continues...
    return NO;
}
```

2. You retrieve a store from the persistent store coordinator, using the URL as an identifier.

```
NSPersistentStoreCoordinator *psc = [[self managedObjectContext]
persistentStoreCoordinator];
NSPersistentStore *pStore = [psc persistentStoreForURL:url];
```

3. If `pStore` is not `nil`, then you can set the metadata. The metadata is a dictionary of key-value pairs, where a key may be either custom for your application, or one of the standard set of Spotlight keys such as `kMDItemKeywords`. Core Data automatically sets values for `NSStoreType` and `NSStoreUUID`, so make a mutable copy of the existing metadata, and then add your own keys and values. In this example, simply set the department name as a keyword, then return `YES`.

   Note that the metadata may be set before validation methods are invoked, so even though the Department name is not optional, although unlikely it is possible for the value to be `nil` at this stage. You should therefore guard against attempting to insert a `nil` value into the array.

```
NSString *departmentName = self.department.name;

if ((pStore != nil) && (departmentName != nil)) {
    NSMutableDictionary *metadata = [[psc metadataForPersistentStore:pStore]
mutableCopy];

    if (metadata == nil) {
        metadata = [NSMutableDictionary dictionary];
    }
```

```
        [metadata setObject:[NSArray arrayWithObject:departmentName]
                forKey:(NSString *)kMDItemKeywords];

        [psc setMetadata:metadata forPersistentStore:pStore];
        return YES;
    }
```

## Complete Code Listing

A complete listing for `setMetadataForStoreAtURL:` is shown in Listing 7-1.

**Listing 7-1** Complete listing of the `setMetadataForStoreAtURL:` method

```
- (BOOL)setMetadataForStoreAtURL:(NSURL *)url {

    NSPersistentStoreCoordinator *psc = [[self managedObjectContext]
persistentStoreCoordinator];
    NSPersistentStore *pStore = [psc persistentStoreForURL:url];
    NSString *departmentName = self.department.name;

    if ((pStore != nil) && (departmentName != nil)) {
       NSMutableDictionary *metadata = [[psc metadataForPersistentStore:pStore]
 mutableCopy];
        if (metadata == nil) {
            metadata = [NSMutableDictionary dictionary];
        }
        [metadata setObject:[NSArray arrayWithObject:departmentName]
                forKey:(NSString *)kMDItemKeywords];
        [psc setMetadata:metadata forPersistentStore:pStore];
        return YES;
    }
    return NO;
}
```

# Set the Metadata for a New Store

When a new store is configured (whether for a new untitled document, or when an existing document is reopened), Core Data calls the `NSPersistentDocument` method `configurePersistentStoreCoordinatorForURL:ofType:modelConfiguration:storeOptions:error:`. You can override this method to add metadata to a new store before it is saved.

## Steps

1. In MyDocument, add an implementation for `configurePersistentStoreCoordinatorForURL:ofType:modelConfiguration:storeOptions:error:`. You first call the superclass's implementation, and check the return value:

```
- (BOOL)configurePersistentStoreCoordinatorForURL:(NSURL *)url
    ofType:(NSString *)fileType
```

```
        modelConfiguration:(NSString *)configuration
        storeOptions:(NSDictionary *)storeOptions
        error:(NSError **)error {

        BOOL ok = [super configurePersistentStoreCoordinatorForURL:url
                    ofType:fileType modelConfiguration:configuration
                    storeOptions:storeOptions error:error];
        if (ok) {
            // implementation continues...
        }
        return ok;
    }
```

2. If the return value for the superclass's implementation is `YES`, then retrieve the persistent store for the specified URL from the persistent store coordinator:

```
NSPersistentStoreCoordinator *psc = [[self managedObjectContext]
persistentStoreCoordinator];
NSPersistentStore *pStore = [psc persistentStoreForURL:url];
```

3. Since the configure method is also called when a document is reopened, you should check for existing custom metadata to avoid overwriting it unnecessarily. If your metadata is not present, set it using `setMetadataForStoreAtURL:`.

```
id existingMetadata = [[psc metadataForPersistentStore:pStore]
        objectForKey:(NSString *)kMDItemKeywords];
if (existingMetadata == nil) {
    ok = [self setMetadataForStoreAtURL:url];
}
```

## Complete Code Listing

A complete listing for `configurePersistentStoreCoordinatorForURL:ofType:modelConfiguration:storeOptions:error:` is shown in Listing 7-2.

**Listing 7-2** Complete listing of the `configurePersistentStoreCoordinatorForURL:ofType:modelConfiguration:storeOptions:error:` method

```
- (BOOL)configurePersistentStoreCoordinatorForURL:(NSURL *)url
    ofType:(NSString *)fileType
    modelConfiguration:(NSString *)configuration
    storeOptions:(NSDictionary *)storeOptions
    error:(NSError **)error {

    BOOL ok = [super configurePersistentStoreCoordinatorForURL:url
                ofType:fileType modelConfiguration:configuration
                storeOptions:storeOptions error:error];
    if (ok) {
        NSPersistentStoreCoordinator *psc = [[self managedObjectContext]
                persistentStoreCoordinator];
        NSPersistentStore *pStore = [psc persistentStoreForURL:url];
        id existingMetadata = [[psc metadataForPersistentStore:pStore]
                objectForKey:(NSString *)kMDItemKeywords];
```

```
        if (existingMetadata == nil) {
            ok = [self setMetadataForStoreAtURL:url];
        }
    }
    return ok;
}
```

# Set the Metadata for an Existing Store

When a document is saved, Core Data calls the `NSPersistentDocument` method `writeToURL:ofType:forSaveOperation:originalContentsURL:error:`. You can override this method to add metadata to the new store before it is saved. (Recall that setting the metadata for a store does not change the information on disk until the store is saved.)

## Steps

1. In MyDocument, add an implementation for `writeToURL: ofType: forSaveOperation: originalContentsURL: error:`. The final step is to invoke and return the superclass's implementation.

```
- (BOOL)writeToURL:(NSURL *)absoluteURL
    ofType:(NSString *)typeName
    forSaveOperation:(NSSaveOperationType)saveOperation
    originalContentsURL:(NSURL *)absoluteOriginalContentsURL
    error:(NSError **)error {

    // implementation continues...

    return [super writeToURL:absoluteURL
                      ofType:typeName
            forSaveOperation:saveOperation
         originalContentsURL:absoluteOriginalContentsURL
                       error:error];
}
```

2. If the document's URL is not `nil`, then it is possible to retrieve the persistent store for that URL from the persistent store coordinator. Invoke `setMetadataForStoreAtURL:` to set the metadata for the store.

```
if ([self fileURL] != nil) {
        [self setMetadataForStoreAtURL:[self fileURL]];
}
```

   Note that this also takes account of Save As operations. The metadata is associated with the persistent store before it is written to a new file.

## Complete Code Listing

A complete listing for `writeToURL:ofType:forSaveOperation:originalContentsURL:error:` is shown in Listing 7-3.

**Listing 7-3** Complete listing of the `writeToURL:ofType:forSaveOperation:originalContentsURL:error:` method

```
- (BOOL)writeToURL:(NSURL *)absoluteURL
    ofType:(NSString *)typeName
    forSaveOperation:(NSSaveOperationType)saveOperation
    originalContentsURL:(NSURL *)absoluteOriginalContentsURL
    error:(NSError **)error {

    if ([self fileURL] != nil) {
        [self setMetadataForStoreAtURL:[self fileURL]];
    }
    return [super writeToURL:absoluteURL
                      ofType:typeName
            forSaveOperation:saveOperation
         originalContentsURL:absoluteOriginalContentsURL
                       error:error];
}
```

# Build and Test

Build and run the application again. Create and save several documents, giving the department a different name in each. Close and then reopen some of the documents, and save some to new locations.

If you open a document in a text editor (such as TextEdit), you should see that the correct metadata is appended to the file.

# What Happened?

You used methods defined by `NSPersistentDocument` to set metadata for a store as it is saved. It is up to you to decide what information to store as metadata, and what keys to specify. You use the keys when writing your importer.

# Writing a Spotlight Importer for Core Data

Details of how in general to write an importer are given in *Spotlight Importer Programming Guide*. This section deals with aspects that are specific to writing an importer for Core Data.

To implement an importer, you first create a new Metadata Importer project in Xcode. Since a Core Data importer uses Objective-C, you should change the file type of the `GetMetadataForFile.c` file from `sourcecode.c.c` to `sourcecode.c.objc` using the Xcode inspector (Info window).

An important aspect of a Spotlight importer is that it should be efficient. A user may have many thousands of files, so any small inefficiency in an importer may have a serious impact on the time it takes to index their disk drive. One of the more expensive tasks in Core Data is creating the persistence stack—the object stores, the object store coordinator, the managed object context, and so on. So that you can avoid this overhead

when reading metadata, `NSPersistentStoreCoordinator` provides a convenience method—`metadataForPersistentStoreWithURL:`—that retrieves the dictionary containing the metadata stored in an on-disk persistent store without initializing a persistence stack.

The main task when you write an importer is to implement the function `GetMetadataForFile`. The function must populate a mutable dictionary—supplied as one of the arguments—with the metadata for the specified file. Given the `NSPersistentStoreCoordinator` convenience method, the code is trivial, as shown here:

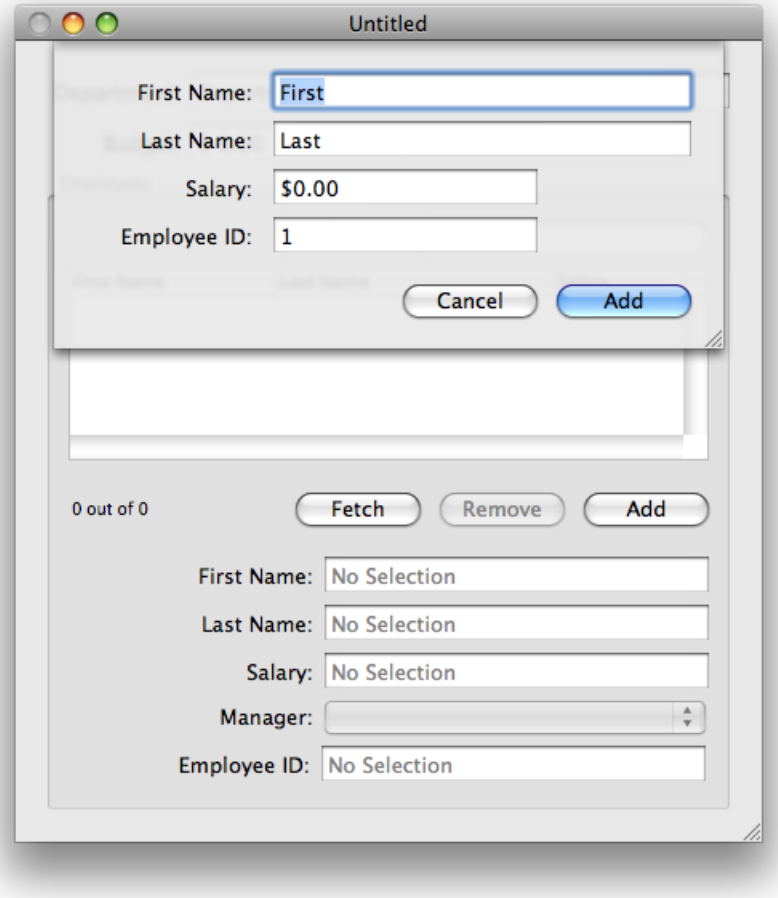
```
Boolean GetMetadataForFile(void* thisInterface,
    CFMutableDictionaryRef attributes,
    CFStringRef contentTypeUTI,
    CFStringRef pathToFile) {

    NSURL *url = [NSURL fileURLWithPath:(NSString *)pathToFile];
    NSDictionary *metadata = [NSPersistentStoreCoordinator
metadataForPersistentStoreWithURL:url error:nil];

    if (metadata != nil) {
        [(NSMutableDictionary *)attributes addEntriesFromDictionary:metadata];
        return TRUE;
    }
    return FALSE;
}
```

In addition to implementing the `GetMetadataForFile` function, you must (as with all importers) modify the CFBundleDocumentTypes entry in the importer project's `Info.plist` file to contain an array of uniform type identifiers (UTIs) for the LSItemContentTypes that your importer can handle, and (if you have defined new attributes) update the `schema.xml` file. These are explained in detail in *Spotlight Importer Programming Guide*.

# A Sheet for Creating a New Employee

It is possible to use the existing application to create new employees. The Add button that is part of the automatically-created user interface is connected to the `add:` method of the Employee array controller. If you click on the button, a new instance of the array controller's entity is added to its content array. Sometimes, though, you might want a more sophisticated user interface. Specifically, some applications use a sheet to allow the user to fill in details about a new entry, and optionally discard the new entry before it is committed. An example of how this could be implemented in the current project is illustrated in Figure 8-1 (page 61).

**Figure 8-1** Creating a new employee using a sheet




In following this section, recall that you are expected to have mastered fundamental Cocoa tools and techniques (see "Introduction to NSPersistentDocument Core Data Tutorial" (page 9))—basic tasks such as connecting outlets and establishing bindings in Interface Builder are not described in detail.

# Design Considerations

There are a number of issues to consider in implementing the sheet. To create a new employee instance, you need a managed object context. A more subtle—but profound—aspect is that you should isolate any changes from the document, so that adding the new employee is a single action that can be undone in a single action. Moreover, it should be possible (if the user clicks Cancel) to discard the new instance without affecting the undo stack in the document.

Together, these constraints suggest a new pattern, where the sheet uses a separate managed object context. The new employee is inserted into this context when the sheet is created. If the user clicks the Add button, the employee is added to the document's main context. If the user clicks the Cancel button, the new employee is deleted and the sheet's context reset—the document's context remains unchanged.

The implementation shown here is but one of several possibilities. Management of the sheet and the managed object context are factored into a separate controller, distinct from the document object. This largely follows the coordinator design pattern (discussed in "Model-View-Controller in Cocoa (Mac OS X)" in *Cocoa Fundamentals Guide*). The goal here, though, is primarily to illustrate the use of Core Data, with comparatively few distractions; it also aims to be fairly reusable. You can adapt what you learn in the tutorial to your own needs. Which approach you take for your own project will depend on the constraints that are specific to your application.

# Implementation Overview

There are several separate steps to the implementation. You need to define the functionality of a new class—the new employee sheet controller—and create a new nib file that contains the sheet.

1. The new employee controller needs to be able to do several things. It must coordinate its own managed object context—which must be configured using the managed object model from the document. The controller must also be able to add the new employee to the document's Department object's employees relationship. To do this, it needs access to that relationship, and again to the document's managed object model.

   The simplest way to satisfy all these needs is to give the new employee controller an outlet to the employees array controller. The array controller gives access to the document's managed object context (to which it is bound), gives the new employee controller an easy way to create a new Employee instance, and (since it manages it) a means to insert the new Employee into the Department's employees relationship.

2. One issue with creating the new top-level object in the nib file is that when you use bindings an object retains other objects to which it is bound. This means that bindings must be broken to ensure there are no retain cycles when a document is closed. Moreover, since the nib file the new controller owns contains top level objects and the controller's class does not inherit from `NSWindowController`, you need to release the top level objects when the window is closed.

3. You need to support undo in the sheet.

# Declaring and Setting up NewObjectSheetController

Create in the project the files for a new class called `NewObjectSheetController`. An instance of this class is responsible for creating and managing the sheet for the new Employee object and for coordinating data between the sheet and the document.

## The header file

The sheet controller uses a private managed object context. It also needs to be able to access the document window, the employees array controller, an object controller in its own nib file, and the sheet. Add to the interface suitable instance variables for all these.

```
@interface NewObjectSheetController : NSObject
{
    NSWindow *documentWindow;
    NSArrayController *sourceArrayController;
    NSObjectController *newObjectController;
    NSPanel *newObjectSheet;

    NSManagedObjectContext *managedObjectContext;
}
```

To complete the interface, add the property and method declarations. The controller needs to provide accessor methods for accessing the managed object contexts, and action methods to launch the sheet, dismiss the sheet, and support undo and redo.

```
@property (nonatomic, retain) IBOutlet NSWindow *documentWindow;
@property (nonatomic, retain) IBOutlet NSArrayController *sourceArrayController;
@property (nonatomic, retain) IBOutlet NSObjectController *newObjectController;
@property (nonatomic, retain) IBOutlet NSPanel *newObjectSheet;

@property (nonatomic, retain, readonly) NSManagedObjectContext
*managedObjectContext;
@property (nonatomic, retain, readonly) NSManagedObjectContext
*documentManagedObjectContext;

- (IBAction)add:(id)sender;

- (IBAction)complete:sender;
- (IBAction)cancelOperation:sender;

- (IBAction)undo:sender;
- (IBAction)redo:sender;

@end
```

## Update the Document nib File

You need to add an instance of the sheet controller in the document nib file and configure it appropriately.

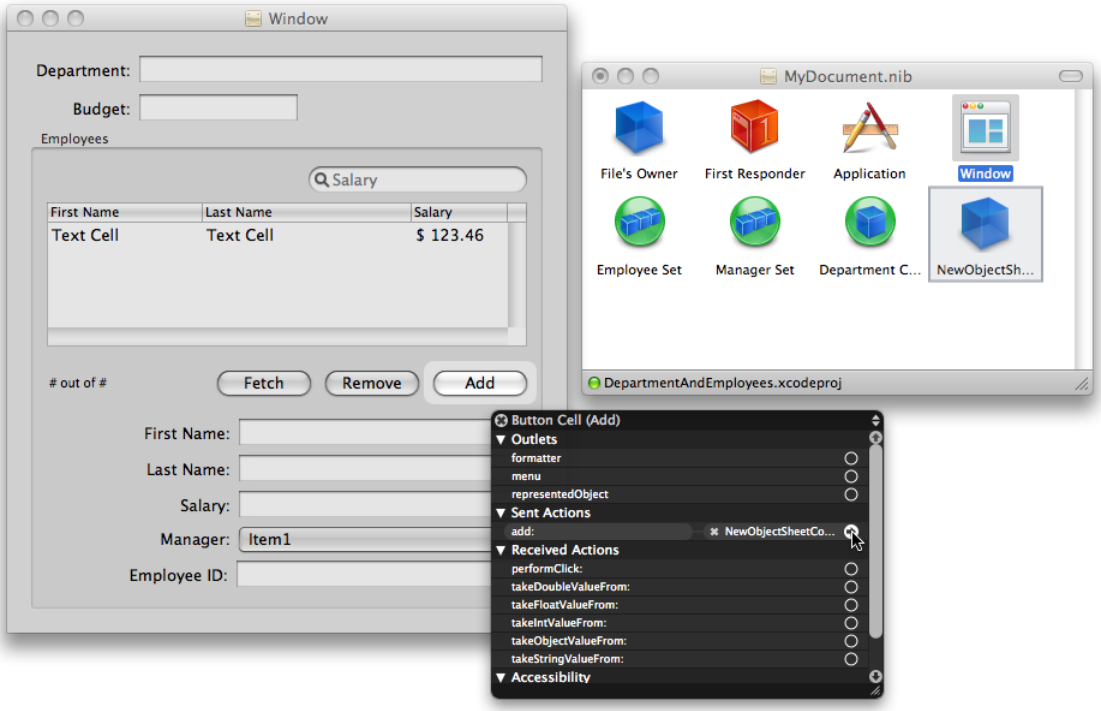
- Import the `NewObjectSheetController` header file into MyDocument.nib, then instantiate an instance.

- Connect the outlets as follows:

  Connect `documentWindow` to the document window.

  Connect `sourceArrayController` to the Employees array controller.

- Now disconnect the Add button from the array controller, and set its target to be the sheet controller and the action to be `add:`.



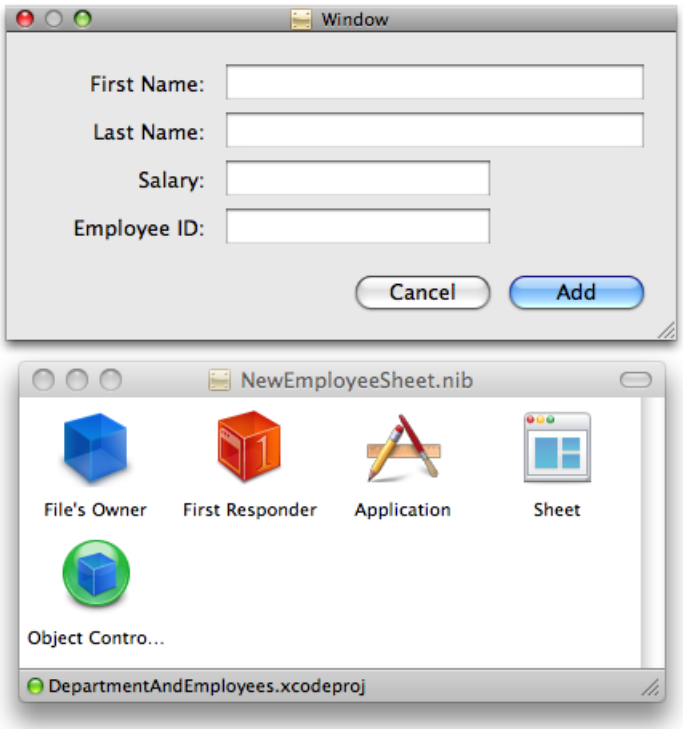

## Create and Configure the Sheet Controller nib File

The next stage is to create and configure the user interface for the sheet.

- Create a new nib file named “NewEmployeeSheet” and add it to the project.
- Import the header file for the `NewObjectSheetController` class into the nib file. Set the File’s Owner to be an instance of `NewObjectSheetController`.
- Add an `NSObjectController` instance to the nib file. Set its entity to be Employee.

  Bind the `managedObjectContext` binding of the `NSObjectController` instance to the `managedObjectContext` key of File’s Owner.
- Connect the File’s Owner’s `newObjectController` outlet to the object controller.
- Add an `NSPanel` instance to the nib file.

  Switch off its Visible at Launch flag.
- Add user interface elements to the panel so that it looks like the image below.




- Connect the File's Owner's `newObjectSheet` outlet to the panel.
- Connect the panel's `delegate` outlet to the File's Owner.
- Bind the values of the text fields to the appropriate attribute in the object controller. For example, bind the value of the First Name text field to the object controller's `selection.firstName` key path. Ensure that the "Validates Immediately" option is selected.
- Set number formatters for the Salary and Employee ID text fields.
- Set up the target and action for the Add and Cancel buttons. The target is File's Owner, the actions are `complete:` and `cancelOperation:` respectively.

# Implement the NewObjectSheetController Class

## Managed Object Contexts

You need access to two different managed object contexts—one from the source array controller (the document's context), and one for the sheet controller itself. You can implement a simple accessor for the source array controller, whereas the sheet controller's accessor lazily creates the context on demand. You set the persistent store coordinator for the controller's context to be the same as that of the document context.

Implement the `managedObjectContext` and `documentManagedObjectContext` methods as follows:

**Listing 8-1** Managed object context accessor methods

```
- (NSManagedObjectContext *)documentManagedObjectContext {
    return [sourceArrayController managedObjectContext];
}

- (NSManagedObjectContext *)managedObjectContext {
    if (managedObjectContext == nil) {
        managedObjectContext = [[NSManagedObjectContext alloc] init];
        [managedObjectContext setPersistentStoreCoordinator:
            [[self documentManagedObjectContext] persistentStoreCoordinator]];
    }
    return managedObjectContext;
}
```

## Setting up the Sheet

You configure the sheet in the `add:` method. The first step is to load the nib file, if necessary.

```
- (IBAction)add:sender {
    if (newObjectSheet == nil) {
        NSBundle *myBundle = [NSBundle bundleForClass:[self class]];
        NSNib *nib = [[NSNib alloc] initWithNibNamed:@"NewEmployeeSheet"
bundle:myBundle];

        BOOL success = [nib instantiateNibWithOwner:self topLevelObjects:nil];
        if (success != YES) {
            // should present error
            return;
        }
    }
    // implementation continues...
}
```

You next need to create a new Employee instance. The easiest way is to use the object controller's `newObject` method. It is also useful to disable undo registration to ensure that the user cannot undo the creation of the new object. This is similar to the initialization of the Department object for a new document.

```
    NSUndoManager *undoManager = [[self managedObjectContext] undoManager];
    [undoManager disableUndoRegistration];

    id newObject = [newObjectController newObject];
    [newObjectController addObject:newObject];

    [[self managedObjectContext] processPendingChanges];
    [undoManager enableUndoRegistration];
```

Finally, you need to display the sheet. Use the `NSApplication` method, `beginSheet:modalForWindow:modalDelegate:didEndSelector:contextInfo:`, and pass `newObjectSheetDidEnd:returnCode:contextInfo:` as the selector.

```
    [NSApp beginSheet:newObjectSheet
            modalForWindow:documentWindow
            modalDelegate:self
          didEndSelector:@selector(newObjectSheetDidEnd:returnCode:contextInfo:)
            contextInfo:NULL];
```

## Responding to Sheet Dismissal

Implement the action methods for the Add and Cancel buttons (see also `complete:` and `cancelOperation:`). Both end the sheet, but with different return codes. Note that the add method also invokes `commitEditing` on the object controller—this ensures that if the user has started editing a field, the pending changes are committed.

**Listing 8-2** Action methods for the Add and Cancel buttons.

```
- (IBAction)complete:sender {
    [newObjectController commitEditing];
    [NSApp endSheet:newObjectSheet returnCode:NSOKButton];
}

- (IBAction)cancelOperation:sender {
    [NSApp endSheet:newObjectSheet returnCode:NSCancelButton];
}
```

When the sheet actually ends, you must respond accordingly in the implementation of `newObjectSheetDidEnd:returnCode:contextInfo:` (the method you defined in "Setting up the Sheet" (page 66) as the callback for the sheet). If the user pressed Add button, you make a new instance of a managed object using the source array controller, and copy the copy attributes from the sheet's managed object.

Whichever button was pressed, you set the content of the sheet's object controller to `nil`, and reset the context (this disposes of all changes). Finally, you order out the sheet.

```
- (void)newObjectSheetDidEnd:(NSWindow *)sheet
    returnCode:(int)returnCode
    contextInfo:(void  *)contextInfo {

    NSManagedObject *sheetObject = [newObjectController content];

    if (returnCode == NSOKButton) {
        NSManagedObject *newObject = [sourceArrayController newObject];
        [newObject setValuesForKeysWithDictionary:
            [sheetObject dictionaryWithValuesForKeys:[[newObject class]
keysToBeCopied]]];
        [sourceArrayController addObject:newObject];
    }

    [newObjectController setContent:nil];
    [[self managedObjectContext] reset];

    [newObjectSheet orderOut:self];
}
```

To suppress compiler warnings for the use of the custom method (`keysToBeCopied`) in the Employee class, you must import the Employee header file. At the top of the implementation file, add:

```
#import "Employee.h"
```

## Tidying Up

To take care of memory management details, the sheet controller must dispose of the sheet and the object controller when the document closes. It must also dispose of the managed object context on `dealloc`. Implement an `awakeFromNib` method to register for window closing notification from the document's window, and implement the corresponding method (`documentWindowWillClose:`) to autorelease `newObjectSheet` and `newObjectController`.

**Listing 8-3** Tidying up methods

```
- (void)awakeFromNib {
    [[NSNotificationCenter defaultCenter] addObserver:self
            selector:@selector(documentWindowWillClose:)
            name:NSWindowWillCloseNotification
            object:documentWindow];
}

- (void)documentWindowWillClose:(NSNotification *)note {
    [[NSNotificationCenter defaultCenter]
        removeObserver:self name:nil object:nil];
    [newObjectSheet autorelease];
    [newObjectController autorelease];
}
```

# Supporting Undo

Although there are only four fields in the sheet, it is still useful to properly support undo—especially if you adapt the techniques described here to a larger task. There are two main steps to this. First, you have to ensure that the sheet uses the sheet controller's managed object context's undo manager and direct undo and redo actions to the undo manager. Second, to ensure the user interface is consistent, you should implement the `validateUserInterfaceItem:` method.

## Accessing the undo manager

First, implement `windowWillReturnUndoManager:` to return the private managed object context's undo manager:

```
- (NSUndoManager *)windowWillReturnUndoManager:(NSWindow *)sender {
    return [[self managedObjectContext] undoManager];
}
```

Next, implement `undo:` and `redo:` methods to direct the actions to the context's undo manager.

```
- (IBAction)undo:sender {
    [[[self managedObjectContext] undoManager] undo];
}

- (IBAction)redo:sender {
    [[[self managedObjectContext] undoManager] redo];
}
```

## Validating user interface items

To validate menu items correctly, you implement the `validateUserInterfaceItem:` method to return the appropriate value. For more details, see *User Interface Validation*.

```
- (BOOL)validateUserInterfaceItem:(id <NSValidatedUserInterfaceItem>)anItem {
    if ([anItem action] == @selector(undo:)) {
        return [[[self managedObjectContext] undoManager] canUndo];
    }
    if ([anItem action] == @selector(redo:)) {
        return [[[self managedObjectContext] undoManager] canRedo];
    }
    return YES;
}
```

# What Happened?

In this section, you followed the coordinator design pattern to create a new controller object that is responsible for managing a sheet used to create a new object. The controller uses a separate managed object context to keep the new managed object, and edits to the new managed object, discrete from the document..

# Document Revision History

This table describes the changes to *NSPersistentDocument Core Data Tutorial*.

| Date | Notes |
| --- | --- |
| 2009-02-04 | Updated description of editing the plist file. |
| 2008-10-15 | Added note that this tutorial targets Mac OS X v10.5. |
| 2008-02-08 | Corrected typographical errors. |
| 2007-12-11 | Updated for Mac OS X v10.5. |
| 2007-08-07 | Corrected a minor formatting error. |
| 2007-07-10 | Added link to completed sample code. |
| 2006-10-03 | Corrected typographical errors. |
| 2006-08-15 | Corrected minor typographical errors. |
| 2006-06-28 | Added a description of using a custom sheet to add a new object. |
| 2006-01-10 | Clarified the steps in the section "Sorting the Managers Popup" and the focus of the error customization section. |
| 2005-12-06 | Corrected typographical errors. |
| 2005-10-04 | Clarified steps in creation of managed object class. |
| 2005-08-11 | Added invocation of super's implementation in awakeFromInsert in Employee class. Added optional section to sort managers pop-up. |
| 2005-07-07 | Corrected minor errors. |
| 2005-04-29 | Corrected the discussion of the cut: method; enhanced the discussion of metadata; enhanced localization section. |
|  | New document that shows how to build a simple but functionally rich application using NSPersistentDocument and Cocoa bindings. |

Document Revision History