

# Chapter 3

## Modeling Tools

Xcode has three modeling tools to help you create Mac and iPhone applications.

- The class modeling tool lets you examine your project's classes, the members of each class, and any relationships between classes.
- The data modeling tool lets you create data structures visually without having to write code.
- The mapping tool lets you create a mapping model to migrate data from an older data model to a new one.

The data modeling tool and the mapping tool require Core Data. If you're not using Core Data, you may want to skip this chapter and come back to it later.

### Class Models

Xcode's class modeling tool is similar to the class browser. Both the class modeling tool and class browser let you examine your project's classes, view the members of each class, and see the inheritance relationships each class has. Why would you want to use the class modeling tool instead of the class browser? The class modeling tool has the following advantages:

- You have more control over what appears in a class model than what appears in the class browser.
- The class modeler draws a class diagram that lets people see your program's structure without having to look at source code.
- You can add comments that explain what each class does.
- You can track the changes you make to a class model in a version control system.

Xcode's class models support Objective-C, C++, and Java programs, but some work is required to use them with Java programs in Xcode 3.2. The class modeling tool needs an Xcode project file. Most Xcode projects have a project file; it has the extension `.xcodproj`. The Java projects that come with Xcode 3.2 do not have a project file so you can't add a class model to them.

The solution for Java developers is to temporarily install Xcode 3.0 or 3.1 in a custom location. Copy the Java project templates to the location of your user templates, which in most cases is in the following location:

```
/Library/Application Support/Developer/Shared/Xcode/Project  
Templates/GroupName
```

Where `GroupName` is the name you want to appear under User Templates on the left side of the New Project Assistant. Java would be a good group name. Read the section “Creating Project Templates” in Chapter 1, “Xcode Projects”, for more information on working with Xcode’s project templates.

## Adding a Class Model to Your Project

To use the class modeler you must create a class model file to store your project’s class models. There are two types of class models you can create: class models and quick models. When you create a class model, Xcode adds it to your project unless you explicitly tell Xcode not to. Class models also let you control what ends up in the model. Quick models add every eligible file to the model. Quick models are initially temporary. You must modify the quick model and save the changes for Xcode to add the quick model to your project.

Choose Design > Class Model > Quick Model to create a quick model file. If you want to add the quick model file to your project, modify the file and save it. You don’t have to do much to modify the file. Looking at a class’s methods and data members in the class diagram modifies the file.

To create a class model:

1. Choose File > New File to create the file.
2. Select Class Model from the file list and click the Next button. The class model file is part of the Other category under Mac OS X.
3. Name the file, select the targets you want to add the class model to, and click the Next button.
4. You will see your project’s Groups and Files list. Add the file groups you want to be in the class model and click the Finish button. The classes in the groups you add are the classes that will appear in the class model. Click the Add All button to add all eligible files to the model.

If you’re writing a Cocoa program and click the Add All button, the class diagram will include all the Cocoa classes. There are a lot of Cocoa classes, which can make finding your classes in the model difficult. If you want to limit the class diagram to classes you wrote, make sure you add only the folders that contain your source code files. For Cocoa Xcode projects, your source code files generally reside in the Classes and Other Sources folders.



## Member List

Selecting a class from the class list fills the member list with the class's members. The member list displays the following information for each member:

- An icon describing the type of member. Member functions have the letter M for method, and data members have the letter V for variable.
- The member name.
- The member kind: method, variable, or class.
- The member's data type. The data type for methods is the data type the method returns.
- The member's visibility: public, private, or protected.
- Whether or not the member has documentation. A book icon signifies that the member has documentation. Clicking the book icon displays the member's documentation in Xcode's documentation window.

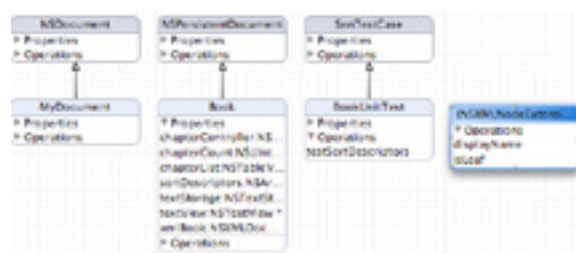
Double-clicking a class member from the member list opens the class's header file.

## Selection Area

Selecting a member from the member list fills the selection area with information about the member. If you select a method, the selection area tells you the name and data type of each argument the method takes.

## Class Diagram

The class diagram, shown in Figure 3.2, contains the classes in the model and the relationships between them. A solid line connecting two classes indicates inheritance, with the line going from the subclass to the superclass. A solid line connecting a class and a category indicates the class that the category is a category of. A dashed line indicates a protocol (or interface for Java programs) implementation.



**Figure 3.2**

Class diagram

Each class in the diagram has three compartments: the class name, properties compartment, and operations compartment. The background color for the class name reflects the kind of class it is. Cocoa classes are gray. Your classes are light blue. Categories are green with the name of the category in parentheses. Protocols are red with the name of the protocol in angled brackets.

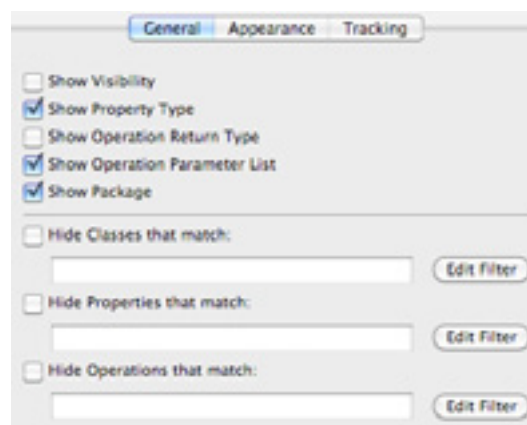
Properties are the data members of the class, and operations are the methods. Initially Xcode shows the class name, properties compartment, and operations compartment, but leaves the class's properties and operations hidden. Click the disclosure triangles to show the properties and operations in their respective compartments. Choosing Design > Roll Up Compartments shows the class name only. Choosing Design > Roll Down Compartments restores the properties and operations compartments.

When you move the mouse cursor over a class property or operation, an arrow appears next to the property or operation. Clicking the arrow opens either the header file (for properties) or the implementation file (for operations in your code). Clicking the arrow for a Cocoa class operation opens the header file.

There are two ways to organize the classes in the diagram: hierarchically and force-directed. Hierarchical layout places parents at the top of the diagram with their children below. Force-directed layout places classes that other classes reference in the center of the graph. A force-directed graph takes longer to create. If you have a lot of classes in your class model, you should use the hierarchical layout. Choose Design > Automatic Layout to choose the layout.

## Opening the Inspector

Opening the inspector for a class model is slightly different than opening other inspectors. Click the diagram without selecting anything. Choose File > Get Info to open the inspector. The inspector should look like Figure 3.3. If the inspector has only the Appearance tab, make sure you didn't select a class or a comment in the diagram before choosing File > Get Info.



**Figure 3.3**

Class model inspector

## Customizing the Class Diagram

To customize the information that appears in the class diagram, open the class diagram's inspector and click the General tab. At the top of the inspector are five checkboxes that control the class member information that appears in the diagram. Selecting the Show Visibility checkbox tells Xcode to show the visibility (public, private, or protected) of each property and operation.

Selecting the Show Property Type checkbox tells Xcode to show the data type of each data member. Selecting the Show Operation Return Type checkbox tells Xcode to show the return type of each class method.

Selecting the Show Operation Parameter List checkbox tells Xcode to show the parameters each method takes. Selecting the Show Package checkbox tells Xcode to show the package that a Java class belongs to.

## Filtering Information from the Diagram

To filter information from the diagram, open the class model's inspector and click the General tab. There are three checkboxes at the bottom of the inspector. Use them to filter classes, methods, and data members from the diagram. Click the Edit Filter button to open the predicate builder, which lets you specify filter conditions.

Click the + button to add a predicate and click the minus button to remove a predicate. Use the left pop-up menu to specify what you're comparing. Use the right pop-up menu to specify the condition. Use the text field to specify the value. If you wanted to filter protocols that start with the letter P:

1. Choose Add AND from the left pop-up menu. Doing so will create two branches out of the predicate.
2. For the top branch, choose Kind from the left pop-up menu.
3. For the top branch, choose = from the right pop-up menu.
4. For the top branch, enter Protocol in the text field.
5. For the bottom branch, choose Name from the left pop-up menu.
6. Choose starts with from the right pop-up menu
7. Enter P in the text field.

## Adding Comments

Xcode class diagrams can have comments that provide an explanation of what a class does. Choose Design > Class Model > Add Comment to create a comment. Select the comment in the class diagram to enter text for the comment. To attach a comment to a class, use the line tool to draw a line from the class to the comment. The line tool is in the lower left corner of the class model window.

## Adding and Removing Files to Track

When you create a class model file, you choose the file groups you want Xcode to track. The classes in the tracked groups are the classes that appear in the class model. When you add source code files to the tracked file groups, the classes in the new source code files automatically appear in the class model. If you forgot to add a file group when you created a class model file, don't worry. You can add file groups after creating the class model file.

To see the file groups Xcode is tracking in the class model, open the inspector and click the Tracking tab. You'll see the groups in your project that Xcode is tracking. To add a group, click the + button. A sheet with your project's groups will open. Select a group from the list and click the Add Tracking button. Selecting a group from the tracking list and clicking the minus button removes the group from the class model.

## Data Models

Xcode's data modeling tool is much more powerful than the class modeling tool. You can actually model data with the data modeling tool, not just view the data. Use the data modeling tool to create your program's data visually instead of writing code.

To use Xcode's data modeling tool, you must be using the Core Data framework. The Core Data framework is available in Mac OS X 10.4 and later and in iPhone OS 3.0 and later. Core Data is a huge topic, too large for me to cover in this book. Read the *Core Data Programming Guide* that comes with the Xcode documentation to learn more about Core Data.

There are three terms you must know before you can model data: entities, attributes, and relationships. Entities are the basic building blocks of your data models. Entities consist of attributes and relationships. Attributes contain data. Relationships represent relationships to other objects. In programming terms, entities are your classes, attributes are your data members, and relationships are your methods.

## Adding a Data Model File to Your Project

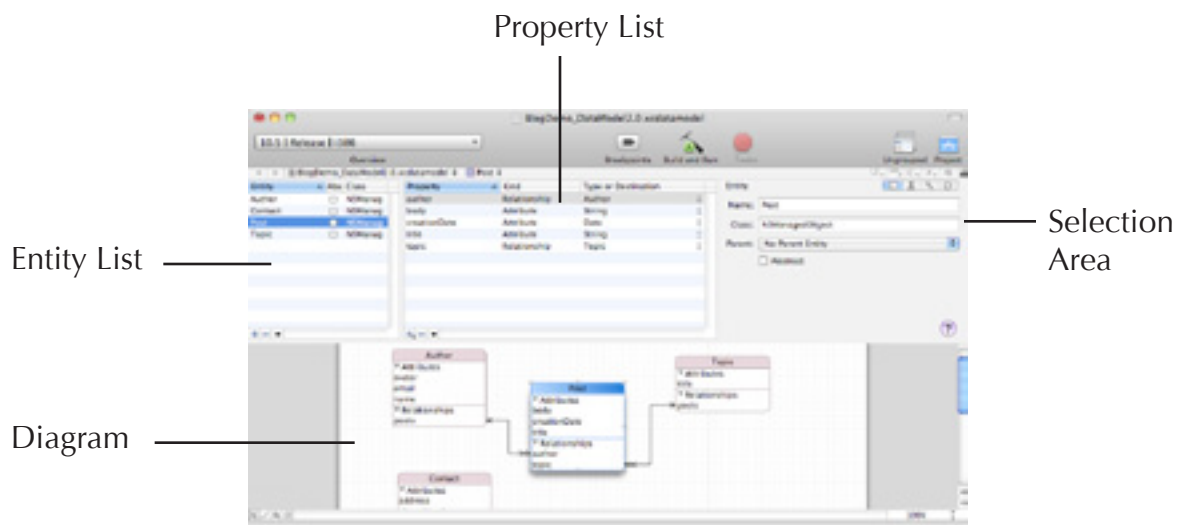
When you create a Core Data application project, Xcode adds a data model file for you. If you have an existing project and want to use the data modeling tool, you must add a data model file to your project.

1. Choose File > New File.
2. Select Data Model from the file list and click the Next button. The data model file is in the Resources group under both Mac OS X and iPhone OS. Choose the appropriate one.
3. Name the file, select the targets you want to add the class model to, and click the Next button. Make sure the data model is added to the application target.
4. Add the classes you want to appear in the data model file and click the Finish button. To add all your project's classes, select the project name and click the Add All button.

## Data Model Window

The data model window, shown in Figure 3.4, is where you examine your data models. The window has four sections.

- Entity list
- Property list
- Selection area
- Diagram



**Figure 3.4**

Data model window

## Entity List

The entity list contains the entities in the data model. For each entity the list tells you the entity's name, the class it belongs to, and whether or not the entity is abstract. The class an entity belongs to must be `NSManagedObject` or a subclass of `NSManagedObject`.

Abstract entities are entities you don't create instances of in your program. They are meant to be inherited by other entities. Cocoa's `NSObject` class is an example of an abstract entity. You don't create objects in your Cocoa programs, you create the entities that inherit from `NSObject`: windows, views, controls, etc.

## Property List

Selecting an entity from the entity list fills the property list with the entity's properties. The property list tells you the property's name and the kind of property it is: attribute or relationship.

## Selection Area

Selecting a property from the property list fills the selection area with information about that property. For an attribute you can set its data type, minimum value, maximum value, and default value. For relationships you can specify the type of relationship and the relationship's destination.

In the upper right corner of the selection area are four buttons, which you can see in Figure 3.5. The buttons, running from left to right, are:

- General, which shows information about the selected property. The General view is the initial view for the selection area.
- User Info, which is used to create information dictionaries.
- Configurations, which is used to create collections of entities.
- Synchronization, which is used to synchronize your data with other applications and devices.

Most of the time, you'll be using the General button. Refer to the "Setting Dictionary Entries" section later in this chapter for more information about the User Info button. Refer to the "Adding Configurations" section later in this chapter for more information on the Configurations button. Refer to the "Synchronizing Data Models" section later in this chapter for more information on the Synchronization button.



**Figure 3.5**

Selection area buttons

## Diagram

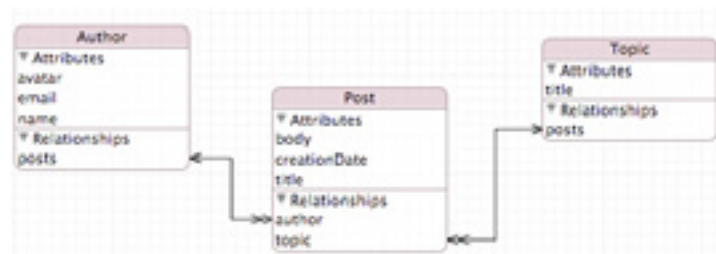
The diagram area, which you can see in Figure 3.6, shows the entities and their relationships to each other. Lines represent the relationships between entities. The arrow points to the destination. Lines with arrowheads on both ends indicate a bidirectional relationship. A line with a single arrowhead indicates a to-one relationship, which means there is only one destination object. A line with a double arrowhead indicates a to-many relationship, which means there can be multiple destination objects.

Each entity in the diagram has at least three compartments: the entity name, attributes compartment, and relationships compartment. If an entity has fetched properties, there will be a fourth compartment for the fetched properties. Initially Xcode shows the class name, attributes compartment, and relationships compartment, but leaves the class's attributes and relationships hidden. Click the disclosure triangles to show the attributes and relationships in their respective compartments. Choosing Design > Roll Up Compartments shows the class name only. Choosing Design > Roll Down Compartments restores the attributes and relationships compartments.

## Adding Entities

There are two ways to add an entity to the diagram. Choose Design > Data Model > Add Entity or click the + button in the entity list. After adding an entity, use the selection area to specify the following information about the entity:

- Its name.
- Its class. The class must be `NSObject` or a subclass of `NSObject`.
- The entity's parent.
- Whether or not the entity is abstract.



**Figure 3.6**

Diagram for data models

When you're starting out, giving each entity the class `NSObject` keeps things simple. Later on you may want to give each entity its own class. Each of these classes will be subclasses of `NSObject`.

Abstract entities are never instantiated. They are meant to be parents of concrete entities.

## Adding Attributes

To add attributes to an entity, select the entity from the diagram or the entity list. Choose `Design > Data Model > Add Attribute` or click the + button in the property list and choose `Add Attribute` from the menu that opens when you click the button.

After adding an attribute, use the selection area to specify the attribute's name and data type. You can also specify whether the attribute is optional and whether it is transient. If an attribute is not optional, it must have a value or you will get errors when the user tries to save the data to disk.

Core Data automatically stores your data in a data file and retrieves the data from the data file. Transient attributes are not stored in the data file.

An attribute can have the following data types:

- Undefined
- Integer 16 (16-bit integer)
- Integer 32 (32-bit integer)
- Integer 64 (64-bit integer)
- Decimal
- Double
- Float
- String
- Boolean
- Date
- Binary Data
- Transformable

Most of the data types correspond to Objective-C data types. The data types that require more explanation are Undefined, Binary Data, and Transformable. Transient attributes should have an undefined data type. Only transient attributes can have an undefined data type.

Give an attribute a data type of binary data when the attribute is more complicated than the Objective-C data types, such as a data structure.

Attributes with a transformable data type are converted to and from instances of `NSData`. Suppose you have an attribute that stores some text you want to display in a text view. You want to display rich text and you want to take advantage of `NSTextView`'s Attributed String binding. If you make the attribute a string, you won't be able to use the Attributed String binding. By making the attribute transformable, you can use the Attributed String binding and display rich text.

When you specify the data type, what you can set depends on the data type you chose. For numerical and date attributes you can set the attribute's minimum, maximum, and default values. For Boolean attributes you can set the default value. For a string you can set the minimum length, maximum length, default value, and a regular expression. Use a regular expression to constrain the values the string can have. If you were using a string attribute to store a number, such as a credit card number, you would use a regular expression to limit the attribute to storing the characters 0–9.

## Adding Relationships

To add a relationship, select the line tool in the bottom left corner of the data model window. Use the line tool to draw a line from the source entity to the destination entity. You can also add a relationship by choosing `Design > Data Model > Add Relationship` or by clicking the + button in the property list and choosing `Add Relationship`.

After adding the relationship, use the selection area to specify the details of the relationship. You can set the following properties for a relationship:

- Name.
- Optional. An optional relationship does not require a destination.
- Transient. Transient relationships are not stored in the data file Core Data uses to store your data.
- Destination. You shouldn't have to change the destination. Choosing `No Destination Entity` from the pop-up menu removes the relationship from the diagram.
- Inverse, which bidirectional relationships use. If you have a relationship from entity A to entity B, the inverse is the corresponding relationship from B to A. Most relationships should have an inverse relationship.
- To-Many. When you have a to-many relationship, the destination can have more than one object. An example of a to-many relationship is a student enrolling in courses. A student can enroll in more than one course.
- Min and Max Counts let you set the minimum and maximum number of destination objects in the relationship.
- Delete rule.

If you don't select the `To-Many Relationship` checkbox, the relationship is a one-to-one relationship. You cannot set the minimum and maximum counts for one-to-one relationships.

Delete rules determine what happens to the source and destination objects of a relationship when you delete the source object. There are four delete rules.

- No Action. Delete the source object but don't delete the destination objects.
- Nullify. Delete the source object. Do not delete destination objects, but set each destination object's inverse relationship to NULL.
- Cascade. Delete the source object and all destination objects.
- Deny. Do not allow the deletion if the source object has destination objects.

## Adding Fetched Properties and Fetch Requests

Fetch properties and fetch requests are two special properties an entity can have. A *fetched property* is a special type of relationship. Supply a set of conditions. The fetched property contains the related objects that meet the conditions you supply. A *fetch request* is an object that tells Core Data the data you want to find. Create a fetch request to retrieve data from an entity.

To add fetch properties and fetch requests to an entity, select the entity from the diagram or the entity list. Choose Design > Data Model > Add Fetch Property to add a fetch property. Choose Design > Data Model > Add Fetch Request to add a fetch request. You can also add fetch properties and fetch requests by selecting an entity from the entity list and clicking the + button in the property list.

## Editing Predicates

Predicates are the way you specify conditions when searching. Fetched properties and fetch requests use predicates. Select a fetch property or fetch request from the browser and click the Edit Predicate button to open the predicate builder.

The predicate builder comes with one predicate for you. If you want to combine multiple conditions, click the + button to add a condition. There are two ways to create a predicate. The first way is to choose Expression from the left pop-up menu. The second pop-up menu disappears. Use the text field to enter the condition.

The second way to create a predicate is to choose one of your attributes from the left pop-up menu. If you're editing a predicate for a fetched property and no attributes appear in the pop-up menu, make sure you choose a destination entity for the fetched property by using the Destination pop-up menu in the selection area.

The second pop-menu up contains conditions. The conditions depend on the attribute's data type. Numerical attributes have mathematical conditions like greater than, less than, and equal. String attributes have additional conditions such as starts with, ends with, and contains. Use the text field to specify the value.

Suppose you have an Employee entity with three attributes: first name, last name, and salary. You want to find the employees whose last names start with J and earn more than \$50,000 a year. To create this predicate:

1. Choose Add AND from the left pop-up menu. Doing so will create two branches out of the predicate.
2. For the top branch, choose last name from the left pop-up menu.
3. For the top branch, choose starts with from the second pop-up menu.
4. For the top branch, enter J in the text field.
5. For the bottom branch, choose salary from the left pop-up menu.
6. Choose greater than from the second pop-up menu.
7. Enter 50000 in the text field.

For more information on predicates, read Apple's *Predicate Programming Guide*. It is part of the Mac OS X and iPhone documentation, which you can read in Xcode.

## Setting Information Dictionary Entries

All elements except fetch requests can have an information dictionary. This dictionary consists of key-value pairs. The information dictionary lets views and controllers access the model's properties. Values used by a fetched property's predicate and version information are examples of data that are commonly placed in information dictionaries.

To modify the information dictionary, click the User Info button in the selection area. The User Info button is the second button in the four button group at the top of the selection area.. Click the + button to add a dictionary entry. Give the entry a key and a value.

## Adding Configurations

A *configuration* is a collection of entities. Configurations let you store entities in different Core Data stores. If Core Data didn't have configurations, you would be limited to one Core Data store that contains the entire data model. Configurations allow multiple stores.

To create a configuration, select an entity from the diagram or the entity list. Click the configuration button in the selection area. The configuration button is the third button in the four button group at the top of the selection area. Click the + button to create a configuration.

After creating the configuration, the next step is to add entities to it. Select an entity and select the checkbox next to the configuration name to add the entity to the configuration.

## Dealing with Entity Changes

Below the configuration list are two text fields that help you handle changes to your data model's entities. The Ver. Hash Modifier text field lets you enter a string that marks the entity as being a new version. Enter a version hash modifier if you make a change to an entity that doesn't change the structure of the data model.

Suppose you had an attribute in your data model measuring customer satisfaction on a scale of 0 to 10. You decide to change the customer satisfaction scale so it ranges from 0 to 100. The structure of the data model is identical in both versions; they both store an integer. But what the data model represents has changed. By entering a version hash modifier, you let Core Data know you have a new version of the data model. You can enter any string you want as a version hash modifier.

The Renaming Identifier text field lets you specify a renaming identifier that resolves naming conflicts. Suppose you have a Customer entity and you change the name to Client. Entering a renaming identifier helps Core Data resolve any conflicts between Customer and Client entities in the data model.

Apple introduced renaming identifiers in Mac OS X 10.6. Don't use them if you want to support earlier versions of Mac OS X.

## Dealing with Property Changes

If you select a property instead of an entity, the selection area changes. There is no list of configurations. However, the Version Hash Modifier and Renaming Identifier text fields remain. These text fields work similarly for properties as they do for entities.

Below the text fields are two checkboxes. If you select the Index in Spotlight checkbox, Core Data makes the property available to Spotlight. When someone using your application enters something in the Finder's search field that matches the property, any files that store the property show up in the search results.

Selecting the Store in External Record file checkbox tells Core Data to store the property in a separate file.

## Synchronizing Data Models

The Synchronizing section is used to sync your data models with other applications and other devices, like laptop computers, cell phones, and iPods. To take advantage of synchronization, you must add the Sync Services framework to your project. Only attributes and relationships can be synchronized.

For more information on synchronizing data, read the *Sync Services Programming Guide*, which is part of Apple's documentation. There is a section on syncing Core Data applications.

### Syncing an Entity

To sync an entity, select it from the entity list and click the Synchronization button in the selection area. The Synchronization button is the last button in the four button group at the top of the selection area. Make sure the Synchronize checkbox is selected.

The last mandatory step is to specify a data class using the Data Class combo box. The data class takes the following form:

```
com.CompanyName.AppName.EntityName
```

If the entity inherits from another entity, choose the parent from the Parent pop-up menu. The Exclude From Data Change Alert checkbox determines what happens when the entity's data changes. If the checkbox is not selected, an alert opens when the entity's data changes. You should limit data change alerts to the most important entities.

### Syncing a Property

To sync a property, select it from the property list and click the Synchronization button in the selection area. Make sure the Synchronize checkbox is selected.

Selecting the Identity Property checkbox tells Core Data to use that property to identify the record. The Exclude From Data Change Alert checkbox works for properties like it does for entities. Deselect it if you want an alert to appear when the property's value changes.

Below the checkboxes are two pop-up menus that are used to automatically resolve conflicts during syncing. The Preferred Client Type menu determines what client Sync Services uses first for the syncing: There are four values:

- App, which is your application.
- Device, which is the physical device, such as a laptop, cell phone, or iPod.
- Server, such as Apple's Mobile Me.
- Peer, which is a peer to peer client.

The Preferred Record menu determines what record should be used for syncing. There are three values:

- Truth, which means use the record in the truth database. The truth database contains all the client's records.
- Client, which means use the record on the client.
- Last Modified, which means use the most recently modified record.

## Creating Source Code

If you create a subclass of `NSObject`, you usually write accessor functions for the class's properties. The data modeling tool can create method declarations and implementations for you. Choose Design > Data Model > Copy Method Declarations to Clipboard to create method declarations. Paste the method declarations in the header file. Choose Design > Data Model > Copy Method Implementations to Clipboard to create method implementations. Paste the implementations in the implementation file.

If you're using Objective-C 2.0, choose Design > Data Model > Copy Obj-C 2.0 Method Declarations to Clipboard to create method declarations. Choose Design > Data Model > Copy Obj-C 2.0 Method Implementations to Clipboard to create method implementations.

## Mapping Models

Mapping models are used to migrate data from one version of a data model to another. They specify the transformations needed to migrate the data. Why would you use a mapping model? Suppose you've written version 1.0 of a Core Data application. Time has passed and you write version 2.0. In the process of writing version 2.0, you made changes to the data model. By using a mapping model, people who saved data using version 1.0 of your application will be able to load their data in version 2.0.

If you're new to Xcode's modeling tools you won't be using mapping models right away because you have to create the initial data model first. When you need to make changes to the initial model, create a mapping model to migrate your data from the initial data model to the new model. Mapping models were introduced in Mac OS X 10.5. If you need to support Mac OS X 10.4, you can't use mapping models. All versions of iPhone OS that support Core Data also support mapping models so there is no reason you can't use mapping models for iPhone applications.

## Adding a New Version of Your Data Model

Before you can use a mapping model, you must create a new version of your existing data model. Select the data model in the project window and choose **Design > Data Model > Add Model Version**. When you add a model version, Xcode converts the data model from a single file to a bundle with the extension `.xcdatamodeld`.

Inside the bundle you will see two versions of the data model: the original and a copy of the data model titled `Filename2.xcdatamodel`, where `Filename` is the name of the original data model. In the project window's detail view, you'll see the original data model has a green check mark next to it. The check mark means the data model is the current version.

Make `Filename2.xcdatamodel` the current version. Select it from the project window and choose **Design > Data Model > Set Current Version**. Open the current data model version (the one with the green check mark). Make the changes you want to make to the current data model. This will involve adding, deleting, or editing the entities, attributes, and relationships in the data model.

## Adding a Mapping Model to Your Project

After making the changes to the data model, you can add a mapping model to your project. Choose **File > New File**. The Mapping Model is in the Resource section under both Mac OS X and iPhone OS. Choose the appropriate one. Click the **Next** button. Name the mapping model and click the **Next** button again.

Now it's time to select the source and destination models for the mapping model. The source model is the model version you didn't modify, the one without the green check mark. The destination model is the model you modified, the one with the green check mark. Remember that the source model is the old version of the data model and the destination model is the new version.

In the New File Assistant you will see a view that looks like the Groups and Files list in Xcode's project window. Underneath the project is a Models folder. Click the disclosure triangle. Your project's data model will have a disclosure triangle next to it. Click it and you will see the two versions of the data model. Select the current version (the one with the green check mark) and click the **Set Destination Model** button. Select the other model and click the **Set Source Model** button. Click the **Finish** button to create the mapping model.

After creating the mapping model, open it to make sure Xcode mapped the data from the old data model to the new data model properly. The mapping should work for relatively simple changes like the following:

- Adding entities, attributes, and relationships.
- Removing entities, attributes, and relationships.
- Renaming entities, attributes, and relationships.
- Making minor changes to an attribute's data type, such as changing an attribute from a 16-bit integer to a 32-bit integer.

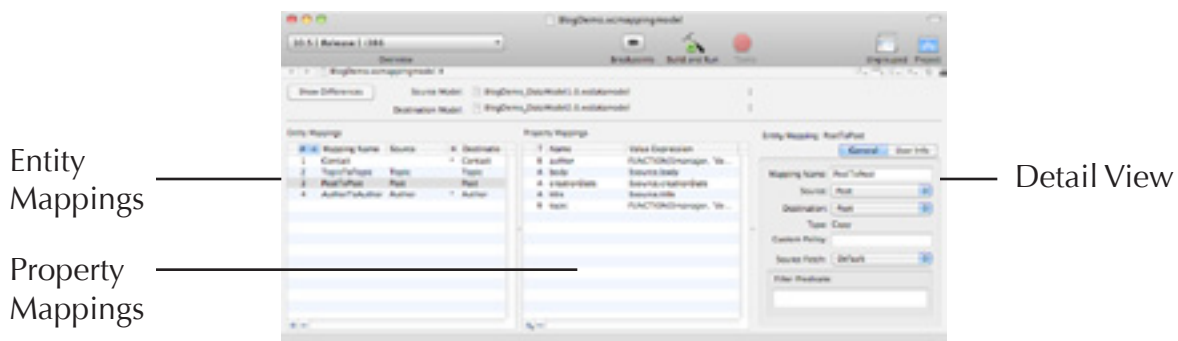
## Mapping Model Window

The mapping model window, shown in Figure 3.7, has four sections. At the top of the window is a button that lets you examine the differences between the source and destination data models. There are also pop-up cells that look like you can change the source and destination data models, but I wasn't able to change them.

### Entity Mappings

Below the top section are the other three sections. The left section is the Entity Mappings section. It has one listing for each entity in the source and destination data models. There are six columns of information.

- Error column, which will be blank if there are no errors. If there is an error, a red X appears in the column.
- Number.
- Mapping Name. If the entity appears in both models, the name will be EntityToEntity, where Entity is the name of the entity. Otherwise, it will be the name of the entity.
- Source, which is the entity name in the source data model.
- H, which is the version hash difference. If the entity is different in the source and destination models, there will be an asterisk in the column.
- Destination, which is the entity name in the destination data model.



**Figure 3.7**

Mapping model window

If there are any errors in your entity mappings, fix them. The data migration won't work properly if your mapping model has errors.

## Property Mappings

Next to the Entity Mappings section is the Property Mappings section. Selecting an entity fills the Property Mappings section with the entity's attributes and relationships. There are four columns of information.

- Error column. If there is an error, a red X appears in this column.
- T, which is the type of property. Attributes have type A, and relationships have type R.
- Name, which is the property's name.
- Value Expression, which represents an expression in a predicate.

New properties have a blank value expression. Attributes in the source data model have the following value expression:

```
$source.AttributeName
```

Relationships in the source model have a value expression that looks similar to the following:

```
FUNCTION($manager, "destinationInstancesForEntityMapping
Named:sourceInstances:", "RelationshipNameTo
RelationshipName", $source.RelationshipName)
```

If there are any errors in your property mappings, fix them. The data migration won't work properly if your mapping model has errors.

## Detail View

The entity mappings and property mapping sections show the mapping data. Use the detail view if you need to change the mapping data. The detail view is on the right side of the mapping model window.

### Changing Entity Mapping Data

Select an entity from the Entity Mappings section to change the entity's mapping data. You can change the mapping's name, source, destination, custom policy, and source fetch.

The custom policy is the name of the class for the mapping. It's a subclass of `NSEntityMigrationPolicy`.

The source fetch determines how Core Data fetches the data from the source data model. There are two source fetch options: default and custom. If you choose the default source fetch, you can supply a filter predicate. A filter predicate is like the predicates you can create for fetched properties and fetch requests in a data model. If you choose a custom source fetch, you can supply a source expression.

### Changing Attribute Mapping Data

If you select an attribute from the Property Mappings section, you can change the value expression. Changing a property's value relationship is optional.

Value expressions can be as complicated as you want to make them, but you should stick to simpler value expressions in the mapping model. If you need a complicated value expression to migrate an attribute, write code to handle the data migration.

A value expression has the data type `NSExpression`. Read the `NSExpression` class reference for more detailed information on value expressions. The `NSExpression` class reference has a link to *Apple's Predicate Programming Guide*, which is also helpful.

### Changing Relationship Mapping Data

If you select a relationship, you can choose whether to auto generate a value expression or use a custom value expression. If you auto-generate the value expression, supply a key path and a mapping name. The key path is one of the relationships in the Property Mappings section. The mapping name is one of the mappings in the Entity Mappings section.

If you use a custom value expression, enter the expression in the Value Expression text field. Value expressions can be as complicated as you want to make them, but you should stick to simpler value expressions in the mapping model. If you need a complicated value expression to migrate a relationship, write code to handle the data migration.

### Creating a User Dictionary

Clicking the User Info tab in the detail view lets you create a dictionary of key/value pairs for a mapping. The information dictionary lets views and controllers access the information contained in the mapping.

## Migrating the Data

After getting the mapping model set, you must write the code to perform the migration. Core Data migration is too large a topic to cover completely here. In this section I explain what you have to do to perform automatic migration. For more information on Core Data migration, read the *Core Data Model Versioning and Data Migration Programming Guide*, which is part of Apple's documentation.

### Enabling Automatic Migration

The first step you must take to migrate your data is to tell Core Data to automatically migrate its persistent stores. Create a variable of type `NSMutableDictionary`. Set the key `NSMigratePersistentStoresAutomaticallyOption` to `YES`.

```
NSMutableDictionary* migrationOptions;
    [migrationOptions setObject:[NSNumber
    numberWithBool:YES]
    forKey:NSMigratePersistentStoresAutomaticallyOption];
```

### Migrating a Document-Based Application

For a document-based application, you must override `NSPersistentDocument's` `configurePersistentStoreCoordinatorForURL` method to migrate the data. Pass the `NSMutableDictionary` variable you created when enabling automatic migration as the `storeOptions` argument.

```
NSMutableDictionary* migrationOptions;
BOOL result = [super
    configurePersistentStoreCoordinatorForURL:url
    ofType:fileType
    modelConfiguration:configuration
    storeOptions:migrationOptions
    error:error];
```

### Migrating a Non-Document Application

When migrating a non-document application, call `NSPersistentStoreCoordinator's` `addPersistentStoreWithType` method. Pass the `NSMutableDictionary` variable you created when enabling automatic migration as the `options` argument. The following code demonstrates migrating an XML store:

## 80 Chapter 3: Modeling Tools

```
NSMutableDictionary* migrationOptions;
NSPersistentStoreCoordinator* newStore;

[newStore addPersistentStoreWithType:NSXMLStoreType
        configuration:nil
        URL:url
        options:migrationOptions
        error:error]
```