
Core Data Model Versioning and Data Migration Programming Guide

[Cocoa > Data Management](#)



2008-02-08



Apple Inc.
© 2008 Apple Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

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

Apple, the Apple logo, Cocoa, Mac, Mac OS, Tiger, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY

DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction to Core Data Model Versioning and Data Migration Programming Guide 7

Organization of This Document 7

Basic Concepts 9

Versioning 11

Concepts 11

Model Versions 13

Mapping Overview 17

Mapping Model Objects 17

Creating a Mapping Model in Xcode 17

The Migration Process 19

Overview 19

Requirements for the Migration Process 19

Custom Entity Migration Policies 20

Three-Stage Migration 20

Initiating the Migration Process 23

Initiating the Migration Process 23

The Default Migration Process 24

Customizing the Migration Process 27

Is Migration Necessary 27

Initializing a Migration Manager 28

Performing a Migration 28

Multiple Passes—Dealing With Large Datasets 29

Document Revision History 31

Figures and Listings

Versioning 11

Figure 1	Recipes models “Version 1.0”	11
Figure 2	Recipes model “Version 1.1”	11
Figure 3	Recipes model “Version 2.0”	12
Figure 4	Initial version of the Core Recipes model	14
Figure 5	Version 2 of the Core Recipes model	15

Mapping Overview 17

Figure 1	Mapping model for versions 1-2 of the Core Recipes models	18
----------	---	----

Initiating the Migration Process 23

Listing 1	Opening a store using automatic migration	24
-----------	---	----

Customizing the Migration Process 27

Listing 2	Checking whether migration is necessary	27
Listing 3	Initializing a Migration Manager	28
Listing 4	Performing a Migration	28

Introduction to Core Data Model Versioning and Data Migration Programming Guide

In Mac OS X v10.5, Core Data provides an architecture to support versioning of managed object models and migration of data from one version to another.

You should read this document if you are an experienced Core Data developer and want to learn how to support versioning in your application.

Important: This document assumes that you are familiar with the Core Data architecture and the fundamentals of using Core Data. You should be able to identify the parts of the Core Data stack and understand the roles of the model, the managed object context, and the persistent store coordinator. You need to know how to create a managed object model, how to create and programmatically interact with parts of the Core Data stack.

If you do not meet these requirements, you should first read the *Core Data Programming Guide* and related materials. You are strongly encouraged also to work through the *Core Data Utility Tutorial*.

Organization of This Document

This document contains the following articles:

- [“Basic Concepts”](#) (page 9) describes the fundamental ideas behind versioning and migration, and outlines the support Core Data provides for these processes.
- [“Versioning”](#) (page 11) describes what is meant by a “version” of a managed object model.
- [“Mapping Overview”](#) (page 17) describes the mapping model.
- [“The Migration Process”](#) (page 19) describes the process of migrating data, including the three stages of migration.
- [“Initiating the Migration Process”](#) (page 23) describes how you start the migration process, and how the default migration process proceeds.
- [“Customizing the Migration Process”](#) (page 27) describes how you can customize the migration process—that is, how you programmatically determine whether migration is necessary; how you find the correct source and destination models and the appropriate mapping model to initialize the migration manager; and then how you perform the migration.

You only customize the migration process if you want to initiate migration yourself. You might do this to, for example, search locations other than the application’s main bundle for models or to deal with large data sets by performing the migration in several passes using different mapping models.

Basic Concepts

This article describes some of the considerations involved in creating different versions of your application, and discusses those aspects for which Core Data provides support.

Typically, as it evolves from one version to another there are numerous aspects of your application that change: the classes you implement, the user interface, the file format, and so on. You need to be aware of and in control of all these aspects; there is no API that solves the problems associated with all these—for example Cocoa does not provide a means to automatically update your user interface if you add a new attribute to an entity in your managed object model. Core Data does not solve all the issues of how you roll out your application. It provides support for a small—but important and non-trivial—subset of the tasks you must perform as your application evolves.

Core Data stores are conceptually bound to the managed object model used to create them. Since a model describes the structure of the data, changing a model will render it incompatible with (and so unable to open) the stores it previously created. If you change your schema, you therefore need to migrate the data in existing stores to new version. In general, managing all this yourself can be difficult.

Core Data provides support for model versioning, for mapping from one model to another, and for data migration. Moreover, it provides an infrastructure to support the process of migration, allowing you to focus on the details of conversion that are specific to your domain.

- Model versioning allows you to specify and distinguish between different configurations of your schema. Core Data also makes it easy to find the right model to open a given persistent store.
- A mapping model parallels a managed object model, specifying how to transform objects in the source into instances appropriate for the destination.
- Data migration allows you to convert data from one model (schema) to another, using mappings.

Although Core Data makes versioning and migration easier than would typically otherwise be the case, it is important to understand that these processes are still non-trivial in effect. You still need to carefully consider the implications of releasing and supporting different versions of your application.

Versioning

This article provides an overview of how versioning is supported in Core Data.

Concepts

There are two distinct views of versioning: your perspective as a developer, and Core Data's perspective. These may not always be the same—consider the following models.

Figure 1 Recipes models "Version 1.0"

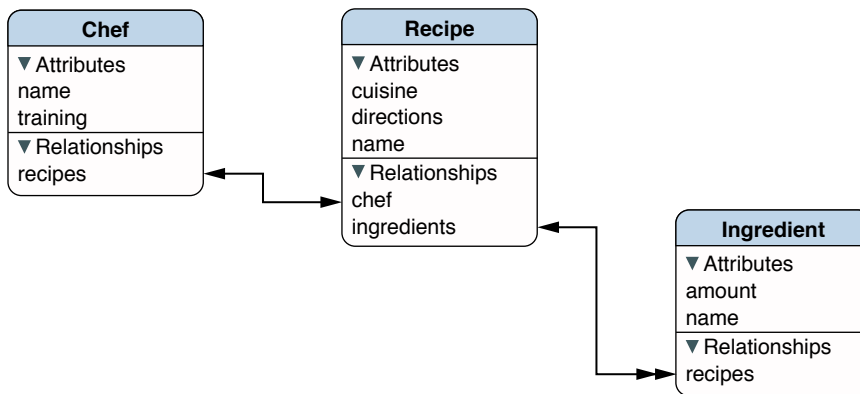


Figure 2 Recipes model "Version 1.1"

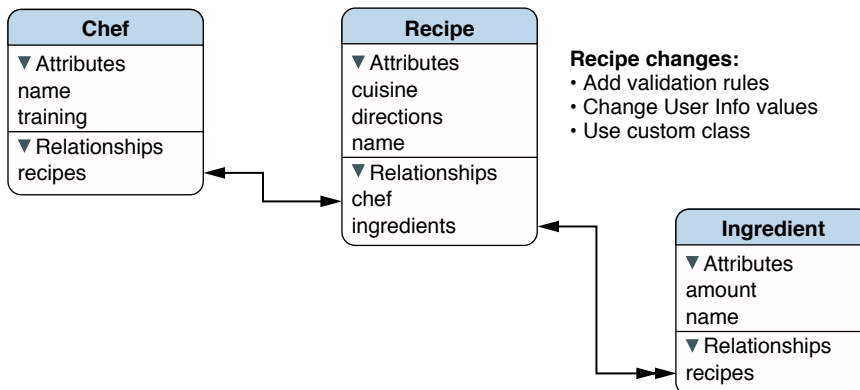
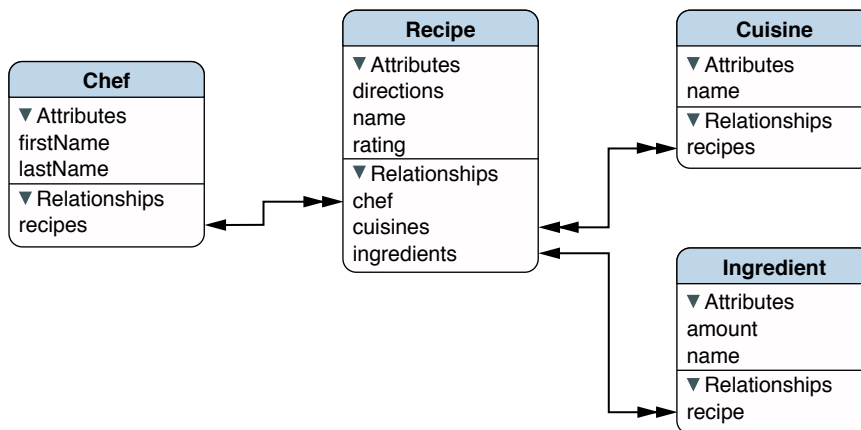


Figure 3 Recipes model “Version 2.0”



As a developer, your perspective is typically that a version is denoted by an identifier—a string or number, such as “9A218”, “2.0.7”, or “Version 1.1”. To support this view, managed object models have a set of identifiers (see `versionIdentifiers`)—typically for a single model you provide a single string (the attribute itself is a set so that if models are merged all the identifiers can be preserved). How the identifier should be interpreted is up to you, whether it represents the version number of the application, the version that was committed prior to going on vacation, or the last submission before it stopped working.

Core Data, on the other hand, treats these identifiers simply as “hints”. To understand why, recall that the format of a persistent store is dependent upon the model used to create it, and that to open a persistent store you must have a model that is compatible with that used to create it. Consider then what would happen if you changed the model but not the identifier—for example, if you kept the identifier the same but removed one entity and added two others. To Core Data, the change in the schema is significant, the fact that the identifier did *not* change is irrelevant.

Core Data’s perspective on versioning is that it is only interested in features of the model that affect persistence. This means that for two models to be compatible:

- For each entity the following attributes must be equal: `name`, `parent`, `isAbstract`, and `properties`.
`className`, `userInfo`, and `validation predicates` are not compared.
- For each property in each entity, the following attributes must be equal: `name`, `isOptional`, `isTransient`, `isReadOnly`, for attributes `attributeType`, and for relationships `destinationEntity`, `minCount`, `maxCount`, `deleteRule`, and `inverseRelationship`.
`userInfo` and `validation predicates` are not compared.

Notice that Core Data ignores any identifiers you set. In the examples above, Core Data treats version 1.0 (Figure 1 (page 11)) and 1.1 (Figure 2 (page 11)) as being compatible.

Rather than enumerating through all the relevant parts of a model, Core Data creates a 32 byte hash digest of the components which it compares for equality (see `versionHash(NSEntityDescription)` and `versionHash(NSPropertyDescription)`). These hashes are included in a store’s metadata so that Core Data can quickly determine whether the store format matches that of the managed object model it may use to try to open the store. (When you attempt to open a store using a given model, Core Data compares the version hashes of each of the entities in the store with those of the entities in the model, and if all are the same then the store is opened.) There is typically no reason for you to be interested in the value of a hash.

There may, however, be some situations in which you have two versions of a model that Core Data would normally treat as equivalent that you want to be recognized as being different. For example, you might change the name of the class used to represent an entity, or more subtly you might keep the model the same but change the internal format of an attribute such as a BLOB—this is irrelevant to Core Data, but it is crucial for the integrity of your data. To support this, Core Data allows you to set a hash modifier for an entity or property see `versionHashModifier (NSEntityDescription)` and `versionHashModifier (NSPropertyDescription)`.

In the examples above, if you wanted to force Core Data to recognize that “Version 1.0” (Figure 1 (page 11)) and “Version 1.1” (Figure 2 (page 11)) of your models are different, you could set (using `setVersionHashModifier:`) an entity modifier for the Recipe entity in the second model to change the version hash Core Data creates .

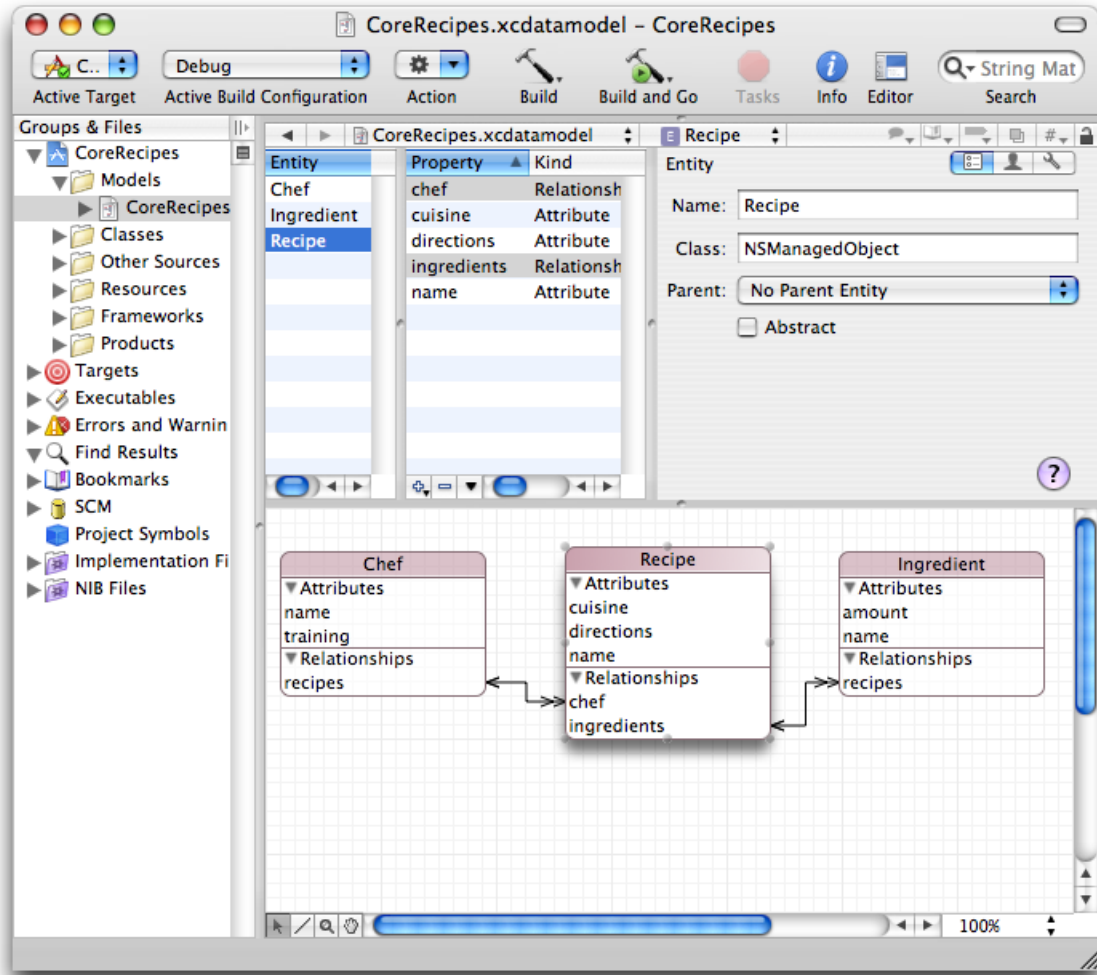
Model Versions

In Mac OS X v10.5, Core Data supports versioned managed object models. The Xcode file type is `.xcdatamodeld` (instead of `.xcdatamodel`) which is a directory that groups versions of a model, each represented by an individual `.xcdatamodel` file, and an `Info.plist` file that contains the version information. Xcode allows you to specify the “current” version.

The versioned model has a new runtime format (`.momd`) that is a bundle containing individually compiled `.mom` files. You load the `.momd` model just as you would a regular `.mom` file (using `NSManagedObjectModel's initWithContentsOfURL:`).

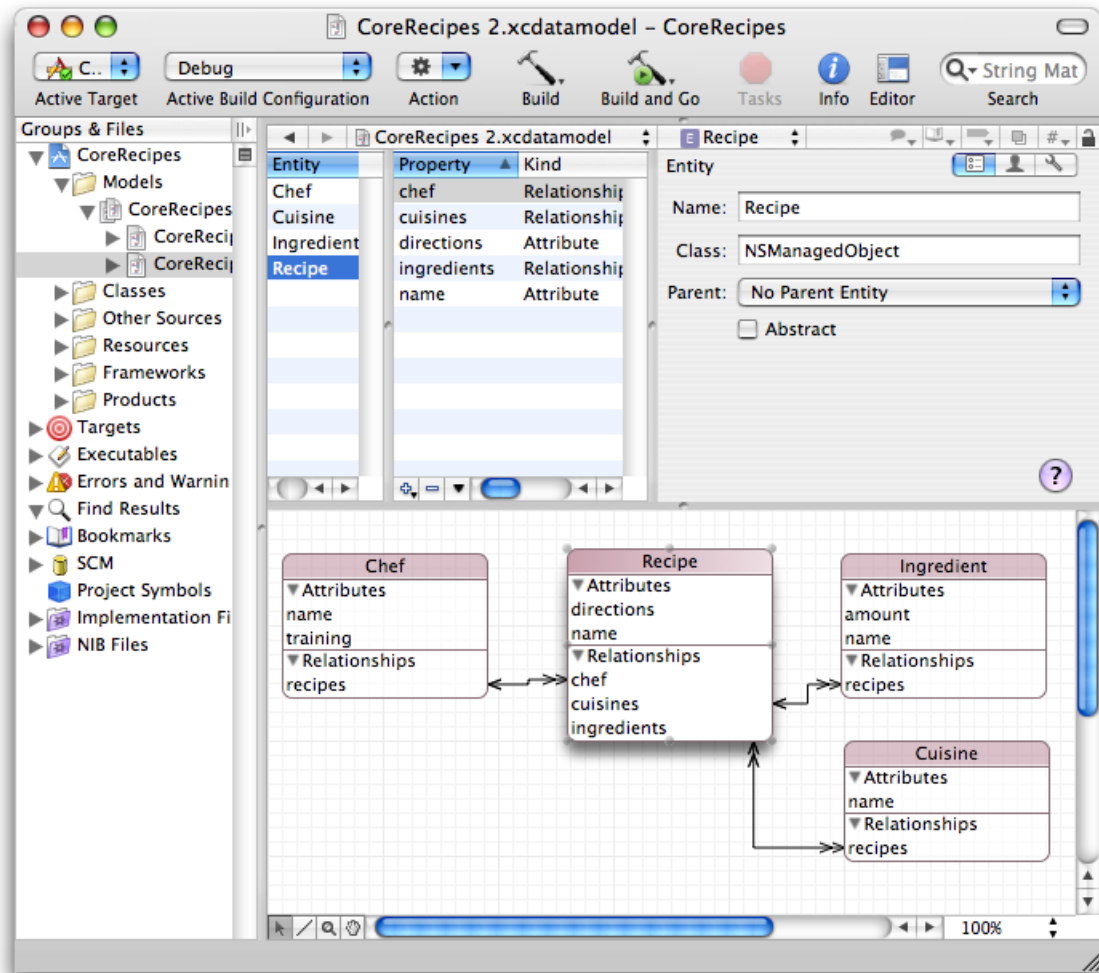
To create a versioned model, you start with a normal model such as that illustrated in Figure 4.

Figure 4 Initial version of the Core Recipes model



To add a version, you select the model in the Groups & Files pane, then select Design > Data Model > Add Model Version. This creates a new directory with the same name as the selected model but with the extension `.xcdatamodeld`, places the original model inside this directory, and makes a copy of the original model as a peer. You can now select the new model and choose Design > Data Model > Set Current Version to denote that it is the current version of the model. You edit the new model just as you would any other model (see Figure 5).

Figure 5 Version 2 of the Core Recipes model



Mapping Overview

This article provides an overview of the mapping model.

In order to transform data from one version of a schema to another, you need a definition of how to perform the transformation. In Core Data, this information is captured in a mapping model. A mapping model is a collection of objects that specifies the transformations that are required to migrate part of a store from one version of your model to another (for example, that one entity is renamed, an attribute is added to another, and a third split into two).

You typically create a mapping model in Xcode. Much as the managed object model editor allows you to graphically create the model, the mapping model editor allows you to customize the mappings between the source and destination entities and properties.

Mapping Model Objects

Like a managed object model, a mapping model is a collection of objects. Mapping model classes parallel the managed object model classes—there are mapping classes for a model, an entity, and a property (`NSMappingModel`, `NSEntityMapping`, and `NSPropertyMapping` respectively).

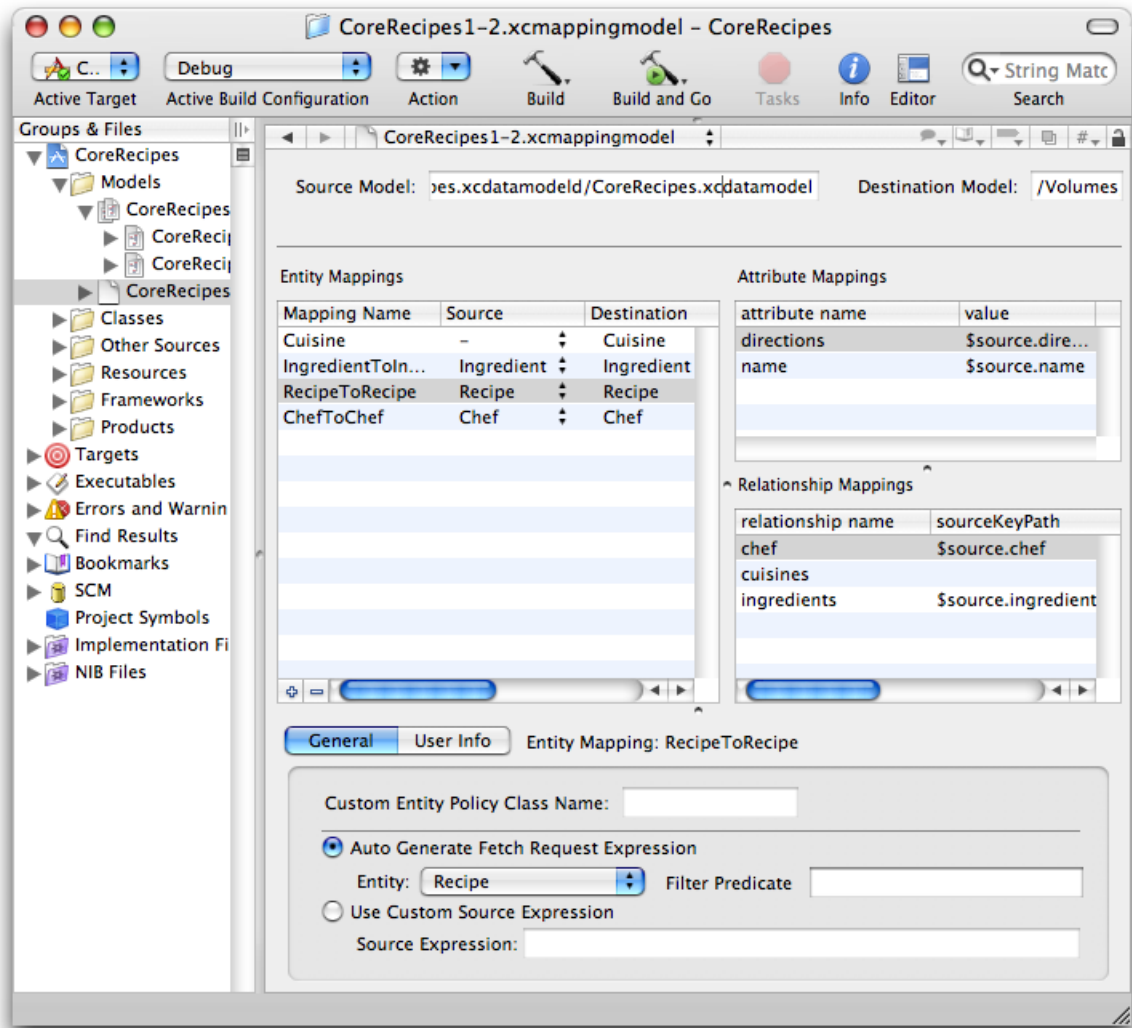
- An instance of `NSEntityMapping` specifies a source entity, a destination entity (the type of object to create to correspond to the source object) and mapping type (add, remove, copy as is, or transform).
- An instance of `NSPropertyMapping` specifies the name of the property in the source and in the destination entity, and a value expression to create the value for the destination property.

The model does not contain instances of `NSEntityMigrationPolicy` or any of its subclasses, however amongst other attributes instance of `NSEntityMapping` can specify the *name* of an entity migration policy class (a subclass of `NSEntityMigrationPolicy`) to use to customize the migration. For more about entity migration policy classes, see [“Custom Entity Migration Policies”](#) (page 20).

Creating a Mapping Model in Xcode

From the File menu, you select New File and in the New File pane select Design > Mapping Model. In the following pane, you select the source and destination models. When you click Finish, Xcode creates a new mapping model that contains as many default mappings as it can deduce from the source and destination. For example, given the model files shown in [Figure 4](#) (page 14) and [Figure 5](#) (page 15), Xcode creates a mapping model as shown in Figure 1.

Figure 1 Mapping model for versions 1-2 of the Core Recipes models



Reserved words in custom value expressions: If you use a custom value expression, you must escape reserved words such as SIZE, FIRST, and LAST using a # (for example, \$source.#size).

The Migration Process

During migration, Core Data creates two stacks, one for the source store and one for the destination store. Core Data then fetches objects from the source stack and inserts the appropriate corresponding objects into the destination stack. Note that Core Data must *re-create* objects in the new stack.

Overview

Recall that stores are bound to their models. Migration is required when the model doesn't match the store. There are two areas where you get default functionality and hooks for customizing the default behavior:

- When detecting version skew and initializing the migration process.
- When performing the migration process.

To perform the migration process requires two Core Data stacks—which are automatically created for you—one for the source store, one for the destination store. The migration process is performed in 3 stages, copying objects from one stack to another.

Requirements for the Migration Process

Migration of a persistent store is performed by an instance of `NSMigrationManager`. To migrate a store, the migration manager requires several things:

- The managed object model for the destination store.
This is the persistent store coordinator's model.
- A managed object model that it can use to open the existing store.
- A mapping model that defines a transformation from the source (the store's) model to the destination model.

You can specify custom entity migration policy classes to customize the migration of individual entities. You specify custom migration policy classes in the mapping model (note the "Custom Entity Policy Name" text field in [Figure 1](#) (page 18)).

Custom Entity Migration Policies

If your new model simply adds properties or entities to your existing model, there may be no need to write any custom code. If the transformation is more complex, however, you might need to create a subclass of `NSEntityMigrationPolicy` to perform the transformation; for example:

- If you have a `Person` entity that also includes address information that you want to split into a separate `Address` entity, but you want to ensure uniqueness of each `Address`.
- If you have an attribute that encodes data in a string format that you want to change to a binary representation.

The methods you override in a custom migration policy correspond to the different phases of the migration process—these are called out in the description of the process given in “Three-Stage Migration.”

Three-Stage Migration

The migration process itself is in three stages. It uses a copy of the source and destination models in which the validation rules are disabled and the class of all entities is changed to `NSManagedObject`.

To perform the migration, Core Data sets up two stacks, one for the source store and one for the destination store. Core Data then processes each entity mapping in the mapping model in turn. It fetches objects of the current entity into the source stack, creates the corresponding objects in the destination stack, then recreates relationships between destination objects in a second stage, before finally applying validation constraints in the final stage.

Before a cycle starts, the entity migration policy responsible for the current entity is sent a `beginEntityMapping:manager:error: message`. You can override this method to perform any initialization the policy requires. The process then proceeds as follows:

1. Create destination instances based on source instances.

At the beginning of this phase, the entity migration policy is sent a `createDestinationInstancesForSourceInstance:entityMapping:manager:error: message`; at the end it is sent a `endInstanceCreationForEntityMapping:manager:error: message`.

In this stage, only attributes (not relationships) are set in the destination objects.

Instances of the source entity are fetched. For each instance, appropriate instances of the destination entity are created (typically there is only one) and their attributes populated (for trivial cases, `name = $source.name`). A record is kept of the instances per entity mapping since this may be useful in the second stage.

2. Recreate relationships.

At the beginning of this phase, the entity migration policy is sent a `createRelationshipsForDestinationInstance:entityMapping:manager:error: message`; at the end it is sent a `endRelationshipCreationForEntityMapping:manager:error: message`.

For each entity mapping (in order), for each destination instance created in the first step any relationships are recreated.

3. Validate and save.

In this phase, the entity migration policy is sent a `performCustomValidationForEntityMapping:manager:error: message`.

Validation rules in the destination model are applied to ensure data integrity and consistency, and then the store is saved.

At the end of the cycle, the entity migration policy is sent an `endEntityMapping:manager:error: message`. You can override this method to perform any clean-up the policy needs to do.

Note that Core Data cannot simply fetch objects into the source stack and insert them into the destination stack, the objects must be re-created in the new stack. Core Data maintains “association tables” which tell it which object in the destination store is the migrated version of which object in the source store, and vice-versa. Moreover, because it doesn't have a means to flush the contexts it is working with, you may accumulate many objects in the migration manager as the migration progresses. If this presents a significant memory overhead and hence gives rise to performance problems, you can customize the process as described in [“Multiple Passes—Dealing With Large Datasets”](#) (page 29).

Initiating the Migration Process

This chapter describes how to initiate the migration process and how the default migration process works. It does not describe customizing the migration process—this is described in [“Customizing the Migration Process”](#) (page 27).

Initiating the Migration Process

When you initialize a persistent store coordinator, you assign to it a managed object model (see `initWithManagedObjectModel:`); the coordinator uses that model to open persistent stores. You open a persistent store using `addPersistentStoreWithType:configuration:URL:options:error:`. How you use this method, however, depends on whether your application uses model versioning and on how you choose to support migration—whether you choose to use the default migration process or custom version skew detection and migration bootstrapping.

- If your application does not support versioning, then you can simply use `addPersistentStoreWithType:configuration:URL:options:error:` directly. If for some reason the coordinator’s model is nevertheless not compatible with the store schema (that is, the version hashes current model’s entities do not equal those in the store’s metadata), the coordinator detects this, generates an error, and `addPersistentStoreWithType:configuration:URL:options:error:` returns NO. You must deal with this error appropriately.
- If your application does support versioning and you choose to use the default migration process, you simply use `addPersistentStoreWithType:configuration:URL:options:error:` as described in [“The Default Migration Process”](#) (page 24). The fundamental difference from the non-versioned approach is that you instruct the coordinator to automatically migrate the store to the current model version by adding an entry to the options dictionary where the key is `NSMigratePersistentStoresAutomaticallyOption` and the value is an `NSNumber` object that represents YES.
- If your application does support versioning and you choose to use custom version skew detection and migration bootstrapping, before opening a store you should check (using `isConfiguration:compatibleWithStoreMetadata:`) whether its schema is compatible with the coordinator’s model: if it is, you can use `addPersistentStoreWithType:configuration:URL:options:error:` to open the store directly; if it is not, you must migrate the store first then open it (again using `addPersistentStoreWithType:configuration:URL:options:error:`).

Note that you could in effect use

`addPersistentStoreWithType:configuration:URL:options:error:` to check whether migration is required, however this is a heavyweight operation and inefficient for this purpose.

It is important to realize that there are two *orthogonal* concepts:

1. You can execute custom code during the migration.

2. You can have custom code for version skew detection and migration bootstrapping.

The migration policy classes allow you to customize the migration of entities and properties in a number of ways, and these are typically all you need. You might, however, use custom skew detection and migration bootstrapping so that you can take control of the migration process. For example, if you have very large stores you could set up a migration manager with the two data models, and then use a series of mapping models to migrate your data into your destination store (if you use the same destination URL for each invocation, Core Data adds new objects to the existing store). This allows the framework (and you) to limit the amount of data in memory during the conversion process.

The Default Migration Process

To open a store and perform migration (if necessary), you simply use `addPersistentStoreWithType:configuration:URL:options:error:` and add to the options dictionary an entry where the key is `NSMigratePersistentStoresAutomaticallyOption` and the value is an `NSNumber` object that represents YES. Your code looks similar to the following example:

Listing 1 Opening a store using automatic migration

```
NSError *error = nil;
NSPersistentStoreCoordinator *psc = /* get a persistent store coordinator */ ;
NSURL *storeURL = /* the URL of a persistent store */ ;
NSDictionary *optionsDictionary =
    [NSDictionary dictionaryWithObject:[NSNumber numberWithInt:YES]
                               forKey:NSMigratePersistentStoresAutomaticallyOption];

BOOL ok = [psc addPersistentStoreWithType:NSSQLiteStoreType
                    configuration:nil
                    URL:storeURL
                    options:optionsDictionary
                    error:&error];
```

If the migration proceeds successfully, the existing store at `storeURL` is renamed with a “~” suffix before any file extension and the migrated store saved to `storeURL`.

In its implementation of `addPersistentStoreWithType:configuration:URL:options:error:` Core Data does the following:

1. Tries to find a managed object model that it can use to open the store.

Core Data searches through your application’s resources for and tests each in turn. If it cannot find a suitable model, Core Data returns NO and a suitable error.

2. Tries to find a mapping model that maps from the managed object model for the existing store to that in use by the persistent store coordinator.

Core Data searches through your application’s resources for available mapping models and tests each in turn. If it cannot find a suitable mapping, Core Data returns NO and a suitable error.

Note that you must have created a suitable mapping model in order for this phase to succeed.

3. Creates instances of the migration policy objects required by the mapping model.

Note that even if you use the default migration process you can customize the migration itself using custom migration policy classes.

Customizing the Migration Process

You only customize the migration process if you want to initiate migration yourself. You might do this to, for example, search locations other than the application's main bundle for models or to deal with large data sets by performing the migration in several passes using different mapping models (see [“Multiple Passes—Dealing With Large Datasets”](#) (page 29)).

Is Migration Necessary

Before you initiate a migration process, you should first determine whether it is necessary. You can check with `NSManagedObjectModel`'s `isConfiguration:compatibleWithStoreMetadata:` as illustrated in [Listing 2](#) (page 27).

Listing 2 Checking whether migration is necessary

```
NSPersistentStoreCoordinator *psc = /* get a coordinator */ ;
NSString *sourceStoreType = /* type for the source store, or nil if not known
*/ ;
NSURL *sourceStoreURL = /* URL for the source store */ ;
NSError *error = nil;

NSDictionary *sourceMetadata =
    [NSPersistentStoreCoordinator metadataForPersistentStoreOfType:sourceStoreType
                                URL:sourceStoreURL
                                error:&error];

if (sourceMetadata == nil) {
    // deal with error
}

NSString *configuration = /* name of configuration, or nil */ ;
NSManagedObjectModel *destinationModel = [psc managedObjectModel];
BOOL pscCompatible = [destinationModel
                    isConfiguration:configuration
                    compatibleWithStoreMetadata:sourceMetadata];

if (pscCompatible) {
    // no need to migrate
}
```

Initializing a Migration Manager

You initialize a migration manager using `initWithSourceModel:destinationModel:`; you therefore first need to find the appropriate model for the store. You get the model for the store using `NSManagedObjectModel`'s `mergedModelFromBundles:forStoreMetadata:`. If this returns a suitable model, you can create the migration manager as illustrated in [Listing 3](#) (page 28) (this code fragment continues from [Listing 2](#) (page 27)).

Listing 3 Initializing a Migration Manager

```
NSArray *bundlesForSourceModel = /* an array of bundles, or nil for the main
bundle */ ;
NSManagedObjectModel *sourceModel =
    [NSManagedObjectModel mergedModelFromBundles:bundlesForSourceModel
                             forStoreMetadata:sourceMetadata];

if (sourceModel == nil) {
    // deal with error
}

MyMigrationManager *migrationManager =
    [[MyMigrationManager alloc]
     initWithSourceModel:sourceModel
     destinationModel:destinationModel];
```

Performing a Migration

You migrate a store using `NSMigrationManager`'s `migrateStoreFromURL:type:options:withMappingModel:toDestinationURL:destinationType:destinationOptions:error:`. To use this method you need to marshal a number of parameters; most are straightforward, the only one that requires some work is the discovery of the appropriate mapping model (which you can retrieve using `NSMappingModel`'s `mappingModelFromBundles:forSourceModel:destinationModel:` method). This is illustrated in [Listing 4](#) (page 28) (a continuation of the example shown in [Listing 3](#) (page 28)).

Listing 4 Performing a Migration

```
NSArray *bundlesForMappingModel = /* an array of bundles, or nil for the main
bundle */ ;
NSError *error = nil;

NSMappingModel *mappingModel =
    [NSMappingModel
     mappingModelFromBundles:bundlesForMappingModel
     forSourceModel:sourceModel
     destinationModel:destinationModel];

if (mappingModel == nil) {
    // deal with the error
}

NSDictionary *sourceStoreOptions = /* options for the source store */ ;
NSURL *destinationStoreURL = /* URL for the destination store */ ;
```

```

NSString *destinationStoreType = /* type for the destination store */ ;
NSDictionary *destinationStoreOptions = /* options for the destination store */
;

BOOL ok = [migrationManager migrateStoreFromURL:sourceStoreURL
           type:sourceStoreType
           options:sourceStoreOptions
           withMappingModel:mappingModel
           toDestinationURL:destinationStoreURL
           destinationType:destinationStoreType
           destinationOptions:destinationStoreOptions
           error:&error];

```

Multiple Passes—Dealing With Large Datasets

The basic approach shown above is to have the migration manager take two models, and then iterate over the steps (mappings) provided in a mapping model to move the data from one side to the next. Because Core Data performs a "three stage" migration—where it creates all of the data first, and then relates the data in a second stage—it must maintain "association tables" (which tell it which object in the destination store is the migrated version of which object in the source store, and vice-versa). Further, because it doesn't have a means to flush the contexts it is working with, it means you'll accumulate many objects in the migration manager as the migration progresses.

In order to address this, the mapping model is given as a parameter of the `migrateStoreFromURL:type:options:withMappingModel:toDestinationURL:destinationType:destinationOptions:error:` call itself. What this means is that if you can segregate parts of your graph (as far as mappings are concerned) and create them in separate mapping models, you could do the following:

1. Get the source and destination data models
2. Create a migration manager with them
3. Find all of your mapping models, and put them into an array (in some defined order, if necessary)
4. Loop through the array, and call `migrateStoreFromURL:type:options:withMappingModel:toDestinationURL:destinationType:destinationOptions:error:` with each of the mappings

This allows you to migrate "chunks" of data at a time, while not pulling in all of the data at once.

From a "tracking/showing progress" point of view, that basically just creates another layer to work from, so you'd be able to determine percentage complete based on number of mapping models to iterate through (and then further on the number of entity mappings in a model you've already gone through).

Document Revision History

This table describes the changes to *Core Data Model Versioning and Data Migration Programming Guide*.

Date	Notes
2008-02-08	Added a note about migrating stores from Mac OS X v10.4 (Tiger).
2007-05-18	New document that describes managed object model versioning and Core Data migration.

