Extracted from:

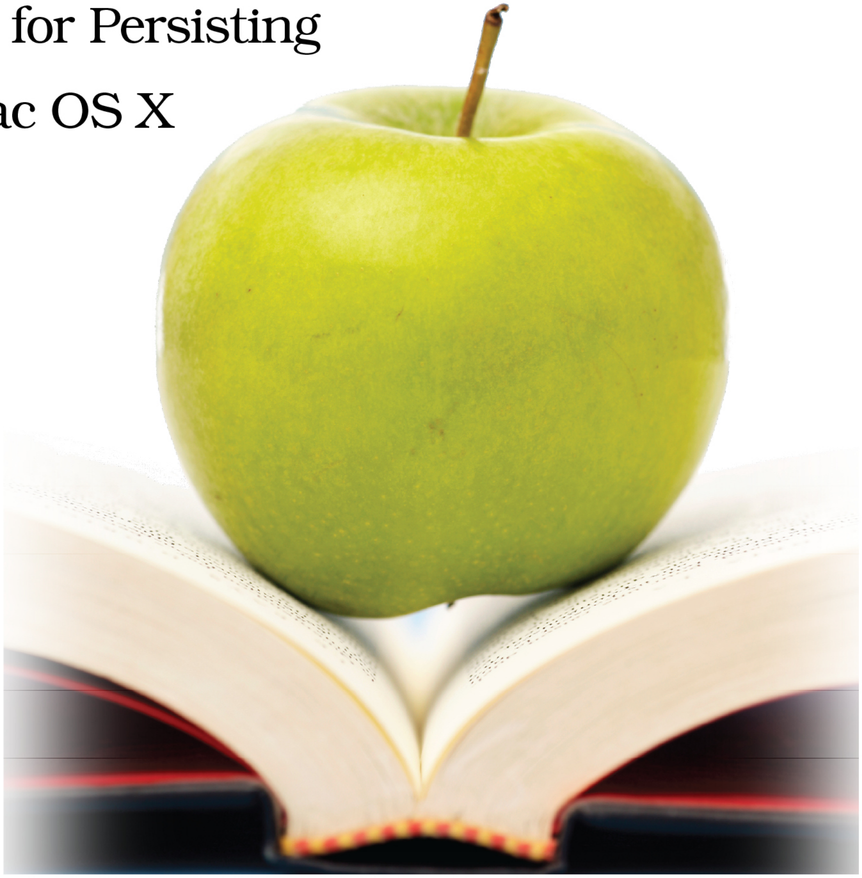
Core Data

Apple's API for Persisting Data on Mac OS X

The Pragmatic Programmers

Core Data

Apple's API for Persisting Data on Mac OS X



Marcus S. Zarra

Edited by Daniel H Steinberg

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragprog.com

Printed in the United States of America.

ISBN-10: 1-934356-32-8

ISBN-13: 978-1-934356-32-6

Printed on acid-free paper.

B6.0 printing, July 29, 2009

Version: 2009-8-25

5.3 Fundamentals of Core Data Versioning

So, what is the magic behind all of this? How does the data migration actually work? As we already explored in the previous chapters, Core Data works with MOM objects that describe the data entities, their attributes, and their relationships. Core Data versioning works with those same MOM objects but takes the design one step further. Each entity version in each data model has a unique hash. When Core Data loads a persistent store from disk, it resolves the matching hashes in the persistent store against the MOM objects included with the application. If the matching MOM is not flagged as the "current" MOM, then data migration kicks in.

How Data Migration Works

Core Data handles data migration in a very straightforward manner. Whenever a persistent store needs to be migrated, there are three steps.

Copying of the Entities with Attributes

In the first pass of the migration, Core Data creates new entities in the new persistent store for every entity in the old store. These entities have their attributes copied over but not their relationships. During this phase, Core Data also keeps a reference to the old unique ID for each entity to be used in phase 2.

Creating Relationships Between the Entities

In the second pass, Core Data builds all the relationships between the entities based on the previous relationships. This is where the reference in phase 1 is used.

Validation of the New Store

During the migration, all validation rules are turned off, and Core Data ignores the child classes defined in the MOM. Therefore, it is possible that some data validation rules got broken during the migration. In the final phase of the migration, Core Data goes back through the store and checks all the validation rules to ensure that the data is in a valid state.

Model Versions and Hashes

The word *versioning* has been used through this chapter as well as other material to describe data migration in Core Data. Unfortunately, it is an inaccurate term. Versioning implies that there is an order or precedence to the models. This is not accurate when it comes to data model versioning/migration in Core Data.

Entity Hashes

Instead of keeping track of a version number, creation date, or some other potentially chronological identifier, Core Data generates a hash for each entity in a model. Those hashes are then stored within the persistent stores created with that model for later comparison. When a persistent store is loaded, the first thing that Core Data does is to retrieve the metadata from that store. Inside that metadata is a list of every entity type in the store along with the hash for that entity. Core Data then compares that list of hashes against the hashes of the "current" MOM. If they match, then everything is fine, and the store is loaded. If they do not match, then Core Data checks the options on the load persistent store call to see whether automatic data migration is requested. If it is not, then the error message from Section 5.1, *Some Maintenance Before We Migrate*, on page 76 is presented to the user.

Changing the Hash Values

Surprisingly, not everything that changes inside a MOM causes the hash of the entities inside to change. There are actually quite a few things that we can do to a model that does not trigger data migration at all.

Changes That Alter the Entity Hash

If any of the following are changed on an entity, the entity will report a different hash:

- Name: The name of the entity
- Inheritance: Changing who the parent entity is
- Persistent properties: Adding or removing a property

In addition, changing the following for properties will also trigger a change to the entity hash:

- Name: The name of the property
- Optionality/read-only: Changing whether the property is optional or read-only
- Attribute type: Changes to the type of value stored

- Relationship: Changes to the destination, minimum/maximum count, the delete rule, or the inverse

Changes That *Do Not* Alter the Entity Hash

The following changes to an entity will *not* trigger a change to the entity hash:

- Class name: Changes to the NSManagedObject subclass
- Transient properties: Properties that are not saved in the persistent store
- User info: Adding, removing, or changing the user info keys/values
- Validation predicates: Adding, removing, or changing the validation rules
- Default values: Adding, removing, or changing the default value of an attribute

In addition, the following changes to the properties of an entity will also *not* change the hash of the entity:

- User info: Adding, removing, or changing the user info key/values
- Validation predicates: Adding, removing, or changing the validation rules

The general distinction between things that do and do not affect version hashes is whether the changes impact the store schema. Things such as the class name impact only the runtime, not the structure of the persistent data.

Mapping Models

If Core Data detects that an upgrade to the persistent store is needed, it looks for three files in the application bundle:

- The MOM that matches the hash from the persistent store
- The current MOM
- The mapping model for those two MOM objects

Assuming that all three files are located (and if they aren't, bad things happen), Core Data will then migrate the data in the persistent store from the old MOM to the new MOM. Once the migration is complete, the stack (MOC, PS, and MOM) is fully initialized, and the application

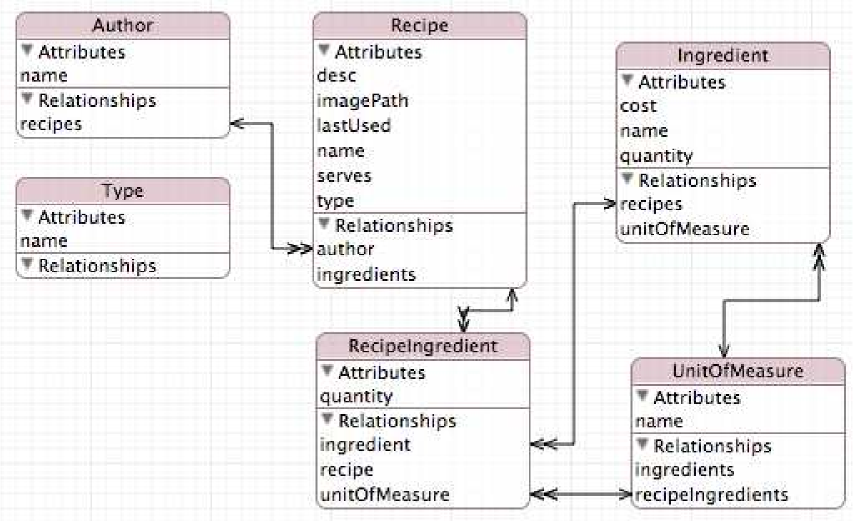


Figure 5.7: Version 3 of the MOM

continues. This, of course, is the happy path, and there are several safeguards in place to allow the application developer to control failures.

5.4 A More Complex Migration

Now that we have gotten our feet wet with data migration and versioning, it's time to test the limits of what we can do. To that end, we will create another migration that is far more complex. Specifically, the ingredients really should be in another entity with a many-to-many relationship to the recipe. In addition, the units of measure should also be in their own table. And since we have the engine apart as it were, we can put in the cost of the ingredients as well as the unit size for ordering. This will allow us to estimate the cost per serving.

With these changes in mind, the data model will look like Figure 5.7. As we learned earlier in this chapter, we will need a mapping model to go from version 2 to version 3. But what about users who are still on version 1? For automatic versioning to work, we would also need a mapping model from version 1 to version 3. Since that will be a variation on our version 2 to version 3 model, we will skip it for the moment.

The biggest challenge for this migration is the introduction of the new entities. Unlike the Author entity from before, during this migration, not only are we creating new entities but we are having to extract data from existing entities to build those new entities, and we have to then properly link the new entities back to their source. To make it even more interesting, we do not want these new entities duplicated. This complexity is far beyond the basic migration that we did for version 2, and it is going to require writing a custom NSEntityMigrationPolicy to handle it.

NSEntityMigrationPolicy

A NSEntityMigrationPolicy allows us to control exactly how a migration is handled. Although there are quite a few methods that we can override depending on our needs, the two methods that we need for this migration are as follows:

```
- (BOOL)createDestinationInstancesForSourceInstance:(NSManagedObject*)source
                                      entityMapping:(NSEntityMapping*)mapping
                                            manager:(NSMigrationManager*)manager
                                              error:(NSError**)error

- (BOOL)createRelationshipsForDestinationInstance:(NSManagedObject*)dInstance
                                    entityMapping:(NSEntityMapping*)mapping
                                          manager:(NSMigrationManager*)manager
                                            error:(NSError**)error
```

createDestinationInstancesForSourceInstance:

The first method, createDestinationInstancesForSourceInstance:, is called for each entity in the source store that is associated with this migration policy. For example, during the migration of the RecipeIngredient entities and the creation of the Ingredient entities, this method would be called for each RecipeIngredient, and it would be expected that an ingredient entity would be created or associated with the incoming RecipeIngredient as a result. The code to implement this breaks down as follows:

Download GrokkingRecipes_v3/RecipeIngredientToIngredient.m

```
NSManagedObjectContext *destMOC = [manager destinationContext];
NSString *destEntityName = [mapping destinationEntityName];

//The name of the ingredient
NSString *name = [source valueForKey:@"name"];
```

In the first part of the method, we are simply setting up references that will be needed later. Specifically, we are getting a reference to the destination NSManagedObjectContext, which we will need to create new entities, the name of the destination entity, and most important the name value from the incoming entity. Since the incoming entity is a RecipeIngredient, the name value will be the name of the ingredient that we now want to reference.

Download GrokkingRecipes_v3/RecipeIngredientToIngredient.m

```
NSMutableDictionary *userInfo = (NSMutableDictionary*)[manager userInfo];
if (!userInfo) {
  userInfo = [NSMutableDictionary dictionary];
  [manager setUserInfo:userInfo];
}
NSMutableDictionary *ingredientLookup = [userInfo valueForKey:@"ingredients"];
if (!ingredientLookup) {
  ingredientLookup = [NSMutableDictionary dictionary];
  [userInfo setValue:ingredientLookup forKey:@"ingredients"];
}
NSManagedObject *dest = [ingredientLookup valueForKey:name];
if (!dest) {
  dest = [NSEntityDescription insertNewObjectForEntityForName:destEntityName
                                       inManagedObjectContext:destMOC];
  [dest setValue:name forKey:@"name"];
  [ingredientLookup setValue:dest forKey:name];
}
```

In this next section of code, we deal with the possibility that the Ingredient entity that we need to reference has already been created. Rather than doing a fetch against the destination context every time, we have a hash built up and stored within the NSMigrationManger. The NSMigrationManager has an NSDictionary called userInfo that is perfectly suited for this purpose. We first lazily initialize this dictionary, and then we lazily initialize another NSDictionary inside it to store references to the Ingredient entities using the name of the ingredient as the key. With this, we can make sure that each Ingredient is created only once. If the Ingredient does not exist yet, then we create it and store it back inside of the userInfo cache.

Download GrokkingRecipes_v3/RecipeIngredientToIngredient.m

```
[manager associateSourceInstance:source
         withDestinationInstance:dest
                forEntityMapping:mapping];

return YES;
```

The last thing that we need to do is to tell the manager about the association. Since the manager keeps track of all associations between the two NSManagedObjectContext objects, we need to inform it of this new entity that was just created and that it is associated with the source entity that was passed in. Once that is complete, we return YES, and we are done.

createRelationshipsForDestinationInstance:

In a properly designed data model, this method will rarely if ever be needed. The intention of this method (which is called in the second pass) is to build any relationships for the new destination entity that was created in the previous method. However, if all the relationships in the model are double sided, then this is not necessary because we already set up one side of them. If for some reason there is an entity in the model that is not double sided, then additional code would be required in this method to handle the one-sided relationship. Since we do not need that functionality in our model, we just return YES.

Download GrokkingRecipes_v3/RecipeIngredientToIngredient.m

```
- (BOOL)createRelationshipsForDestinationInstance:(NSManagedObject*)dInstance
                                    entityMapping:(NSEntityMapping*)mapping
                                          manager:(NSMigrationManager*)manager
                                            error:(NSError**)error
{
  return YES;
}
```

5.5 Automatic Data Migration

If your data migration needs are easy to handle and your application is not coming from Tiger, then automatic migration is probably all that is needed. Automatic migration lets Core Data handle all the details and assumes the following:

- Every persistent store that the application will come up against has hash metadata.
- Every persistent store that the application will come up against has a corresponding model stored inside the application's bundle.
- Every persistent store that the application will come up against has a mapping model from its MOM to the current MOM.

If the application can meet these three criteria (and any application that has begun its life in Leopard should), then automatic migration should be able to do all of the dirty work for us.

To enable automatic versioning, we need to set a preference on the NSPersistentStoreCoordinator while adding a persistent store. Previously,

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Core Data's Home Page
http://pragprog.com/titles/mzcd
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/mzcd.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	`support@pragprog.com`
Non-English Versions:	`translations@pragprog.com`
Pragmatic Teaching:	`academic@pragprog.com`
Author Proposals:	`proposals@pragprog.com`
Contact us:	1-800-699-PROG (+1 919 847 3884)