Mobile Application Development

Lecture 22
Core Data

Lecture Summary

- Core Data
- NSManagedObjectContext
- UIManagedDocument
- NSManagedObject: key-value coding and subclassing
- Inserting and deleting objects
- Fetching objects
- NSFetchedResultsController and UITableView
- DEMO

Core Data

- The Core Data framework (`#import <CoreData/CoreData.h>`) provides generalized and automated solutions to common tasks associated with object life-cycle and object graph management, including persistence
- Core Data provides support for:
  - change tracking and undo operations
  - relationship maintenance
  - smart (lazy) loading of objects
  - validation of property values
  - integration with the application's controller to synchronize UI
  - key-value coding and key-value observing
  - complex querying on object graphs

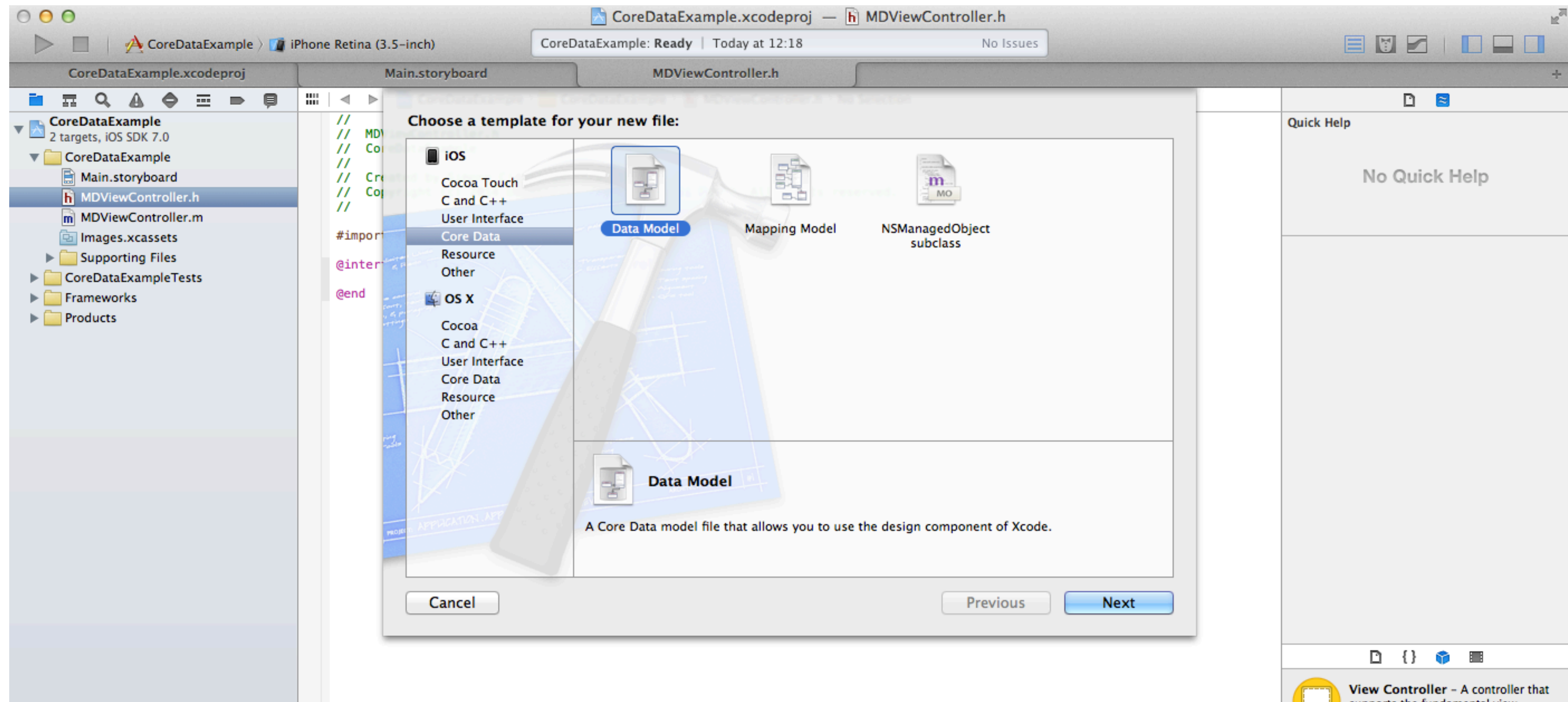
https://developer.apple.com/library/ios/documentation/cocoa/conceptual/CoreData/cdProgrammingGuide.html

Core Data

- iOS applications often need to store and retrieve (query) data
- Data to be stored are objects (in code they will be of class `NSManagedObject`)
- Core Data provides an object-oriented database which stores an object graph and can query for objects efficiently and in an object-oriented fashion
- Persistence of object graphs can be handled by Core Data in different ways:
  - SQLite
  - XML
  - In-Memory
  - Atomic

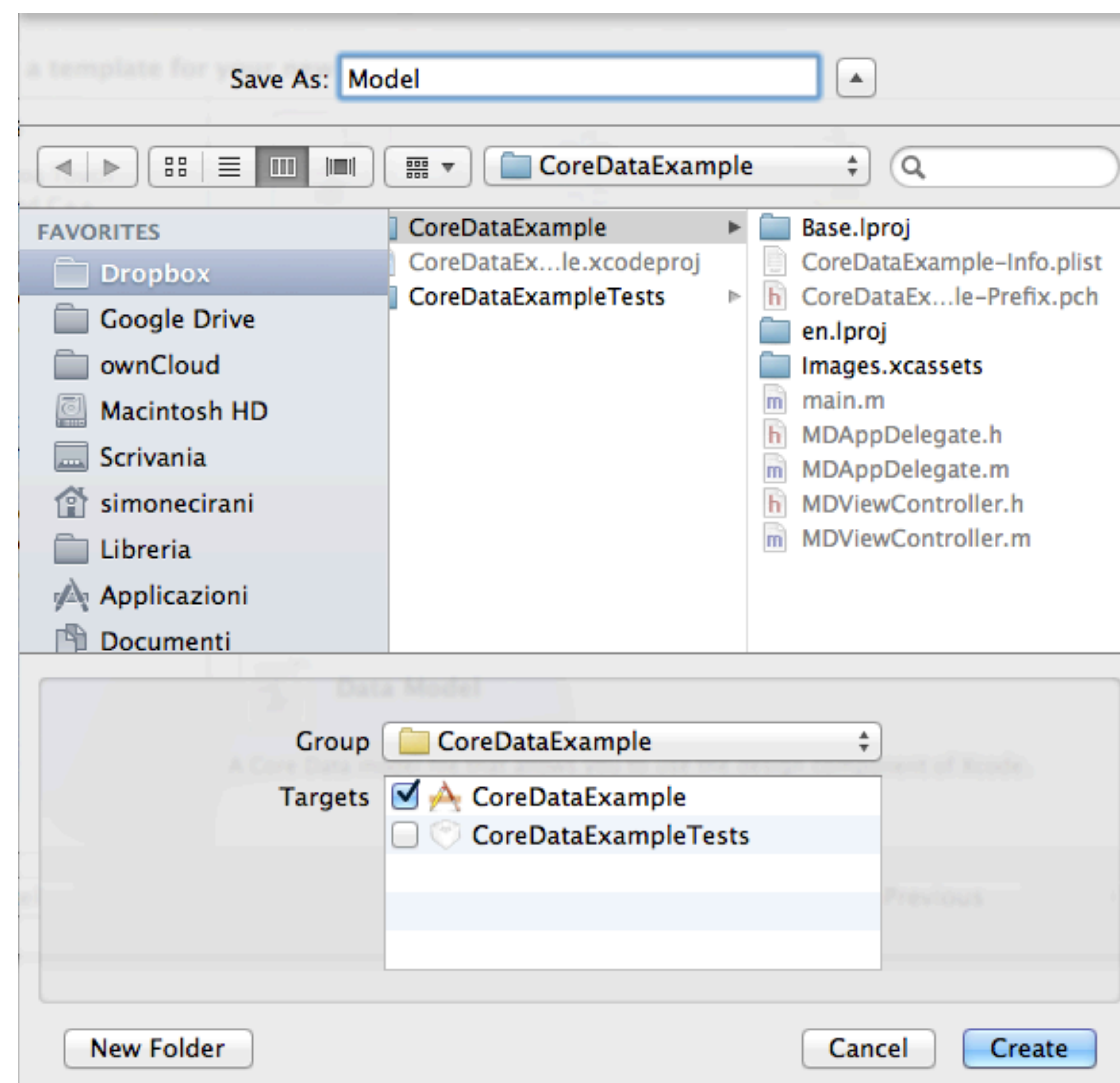
Core Data

- Xcode provides a visual tool to map objects with the underlying database
- To do so, create a new Data Model file in the project



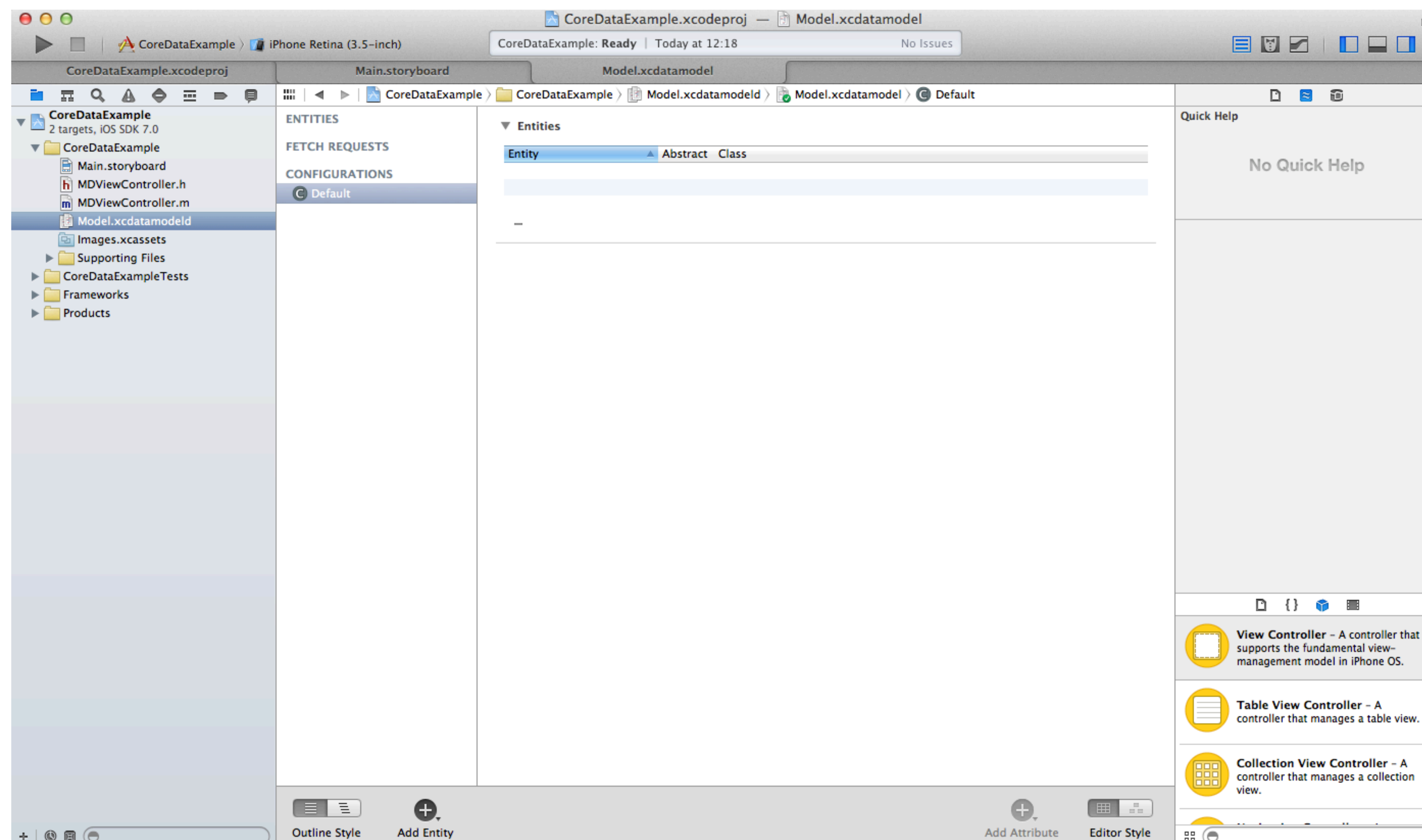
Core Data

- Give a name to the Data Model and click "Create"



Core Data

- The Data Model file has **.xcdatamodel** extension
- Clicking on the Data Model file will open the visual tool to define the object graph



Core Data

- A managed object model is an instance of the `NSManagedObjectModel` class
- The Data Model in composed by
  - **Entities** are model objects that encapsulate specified data and provide methods that operate on that data; entities may be arranged in an inheritance hierarchy may be specified as abstract; <u>you can think of entities as classes</u>
  - **Attributes** are structures that contain data; attributes can have a scalar value (`int`, `double`, `float`, `char`, C structs) or be instances of primitive classes (`NSString`, `NSNumber`, `NSData`, ...); <u>you can think of attributes as properties</u>
  - **Relationships**: represents the arcs in the graph (the relations among objects); relationships are bidirectional; relationships can be "to-one" (if the destination is a single object) or "to-many" (if the destinations are multiple objects); relationships can be mandatory or optional and can have a range for the cardinality (e.g., [0,1], [1,10], [1,n], ...); <u>you can think of relationships as pointers to other objects</u>

Core Data

Entities

CoreDataExample 〉 CoreDataExample 〉 Model 〉 Model.xcdatamodeld 〉 Model.xcdatamodel 〉 Entity

ENTITIES
Entity
FETCH REQUESTS
CONFIGURATIONS
Default

▼ Attributes

Attribute ▲	Type

\+ −

▼ Relationships

Relationship ▲	Destination	Inverse

\+ −

▼ Fetched Properties

Fetched Property ▲	Predicate

\+ −

Outline Style   Add Entity   Add Attribute   Editor Style

Core Data

CoreDataExample 〉 CoreDataExample 〉 Model 〉 Model.xcdatamodeld 〉 Model.xcdatamodel 〉 Entity

ENTITIES
Entity
FETCH REQUESTS
CONFIGURATIONS
Default

Attributes

Attribute ▲	Type

Attributes

+ −

Relationships

Relationship ▲	Destination	Inverse

+ −

Fetched Properties

Fetched Property ▲	Predicate

+ −

Outline Style · Add Entity · Add Attribute · Editor Style

Core Data

CoreDataExample 〉 CoreDataExample 〉 Model 〉 Model.xcdatamodeld 〉 Model.xcdatamodel 〉 Entity

ENTITIES
Entity
FETCH REQUESTS
CONFIGURATIONS
Default

Attributes

Attribute ▲	Type

Relationships

Relationship ▲	Destination	Inverse

Fetched Properties

Fetched Property ▲	Predicate

Outline Style · Add Entity · Add Attribute · Editor Style

Relationships

Adding an Entity to the Data Model

ENTITIES
FETCH REQUESTS
CONFIGURATIONS

▼ Attributes

Attribute ▲ Type

+ −

▼ Relationships

Relationship ▲ Destination Inverse

+ −

▼ Fetched Properties

Fetched Property ▲ Predicate

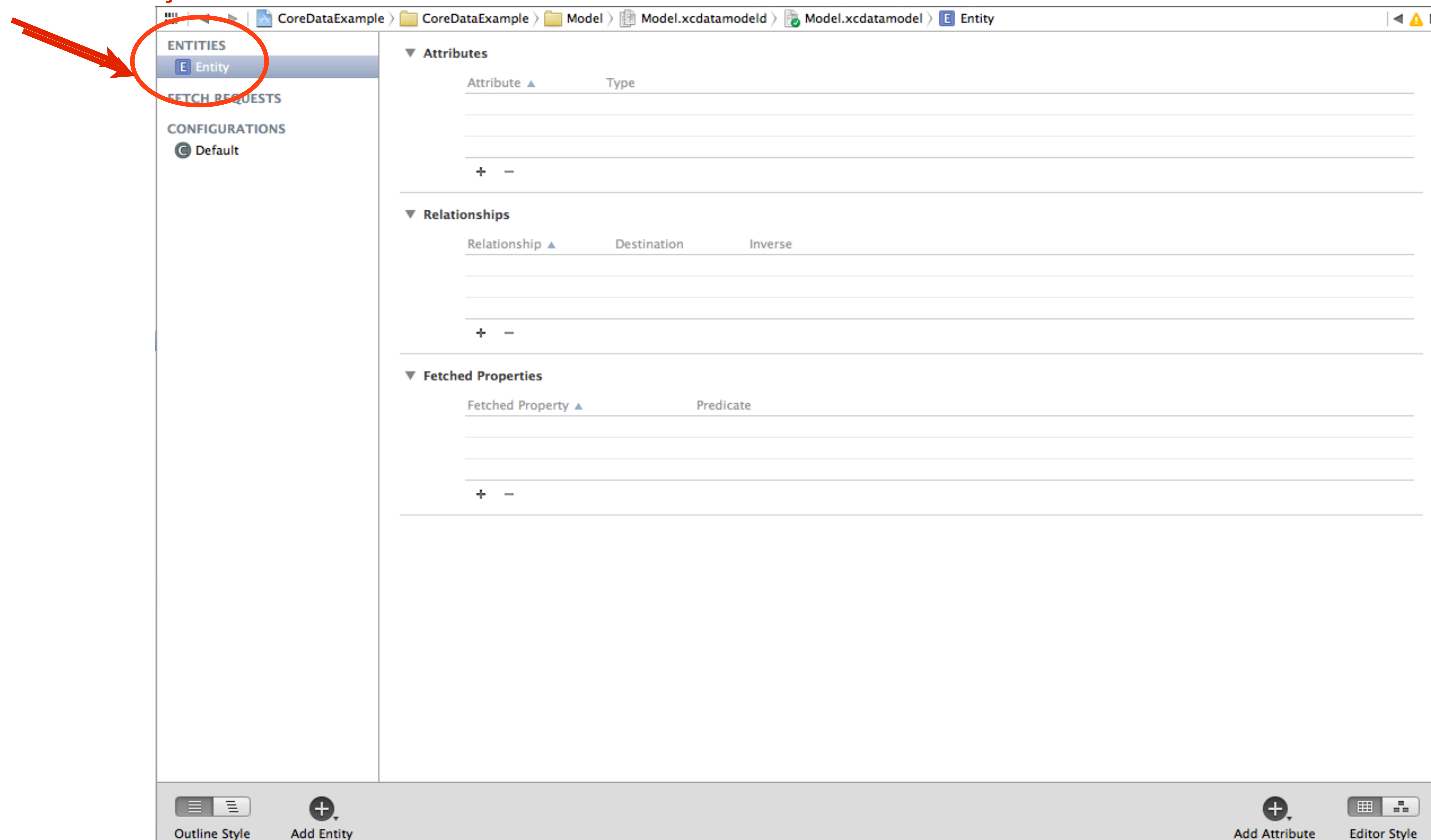
+ −

Outline Style
Add Entity
Add Attribute
Editor Style

Open the Data Model file and click on “Add Entity”

Adding an Entity to the Data Model

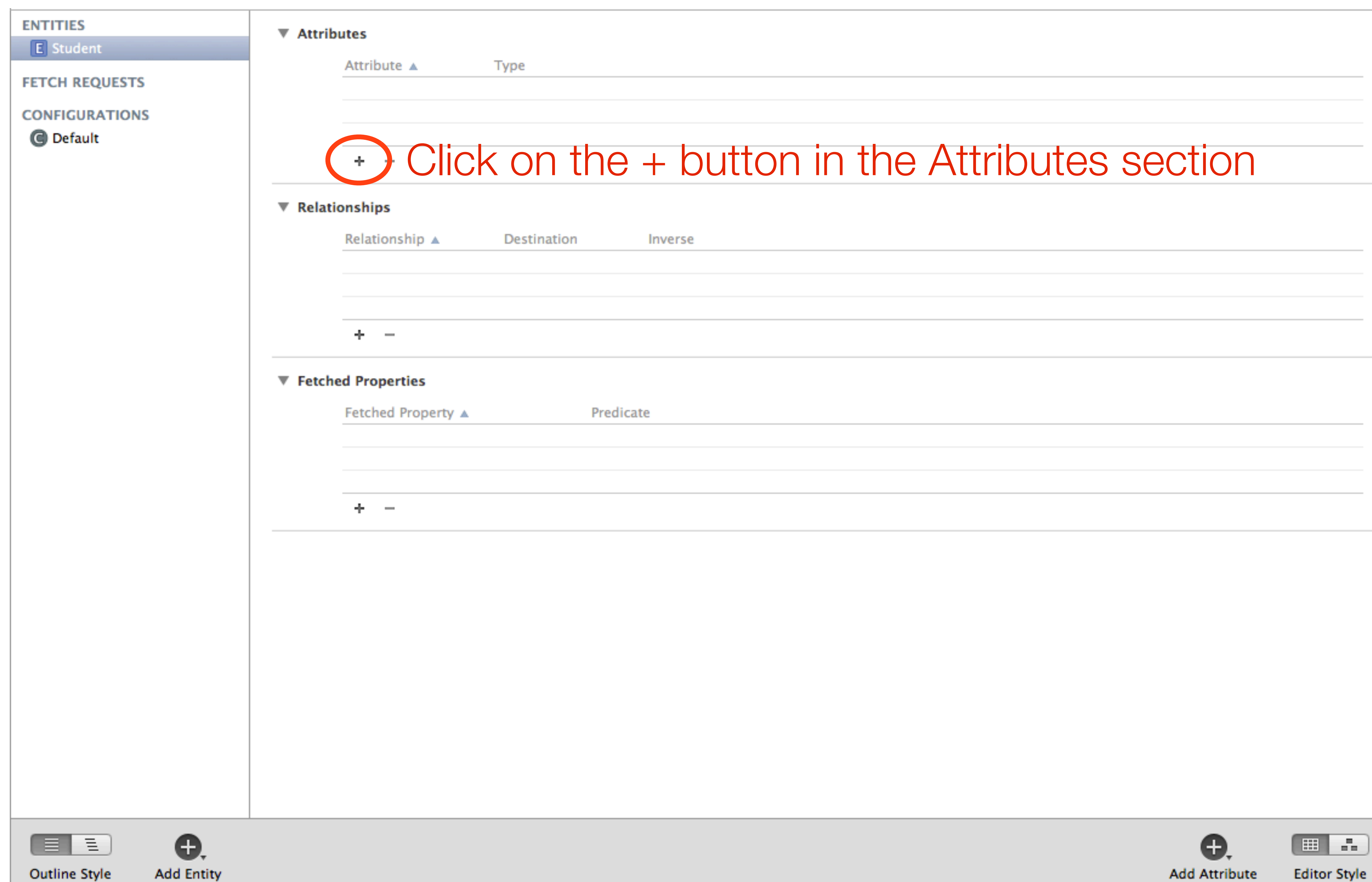
The new entity is added to the model



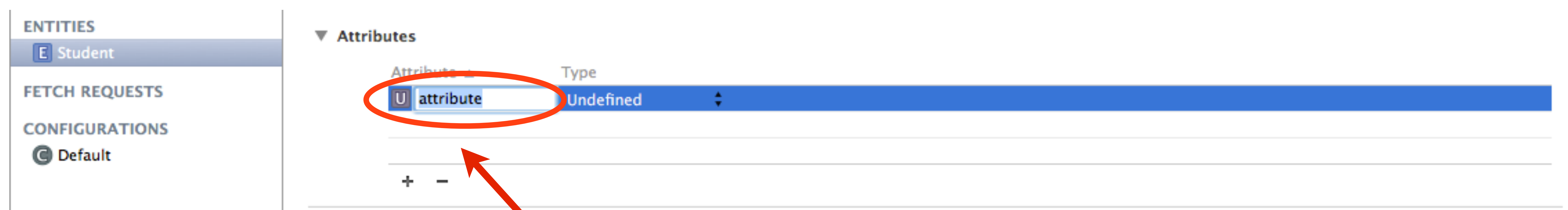
Adding an Entity to the Data Model

The entity's name can be changed by double-clicking and typing

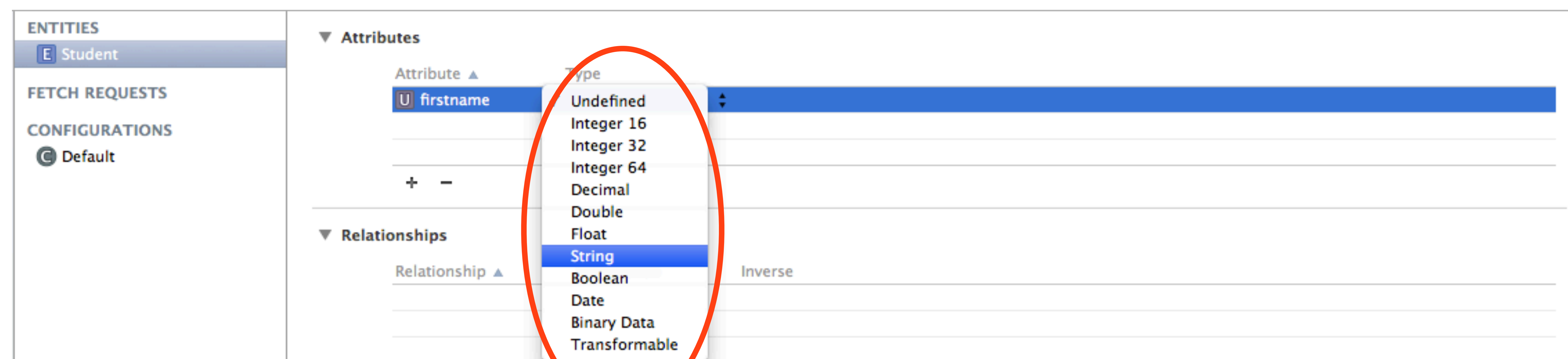
Adding an Attributes to an Entity



Adding an Attributes to an Entity



Type the attribute's name



Select the attribute's type

Model graph view

ENTITIES
Course
Student
Teacher
FETCH REQUESTS
CONFIGURATIONS
Default

Course
Attributes
endDate
name
startDate
Relationships

Student
Attributes
firstname
lastname
Relationships

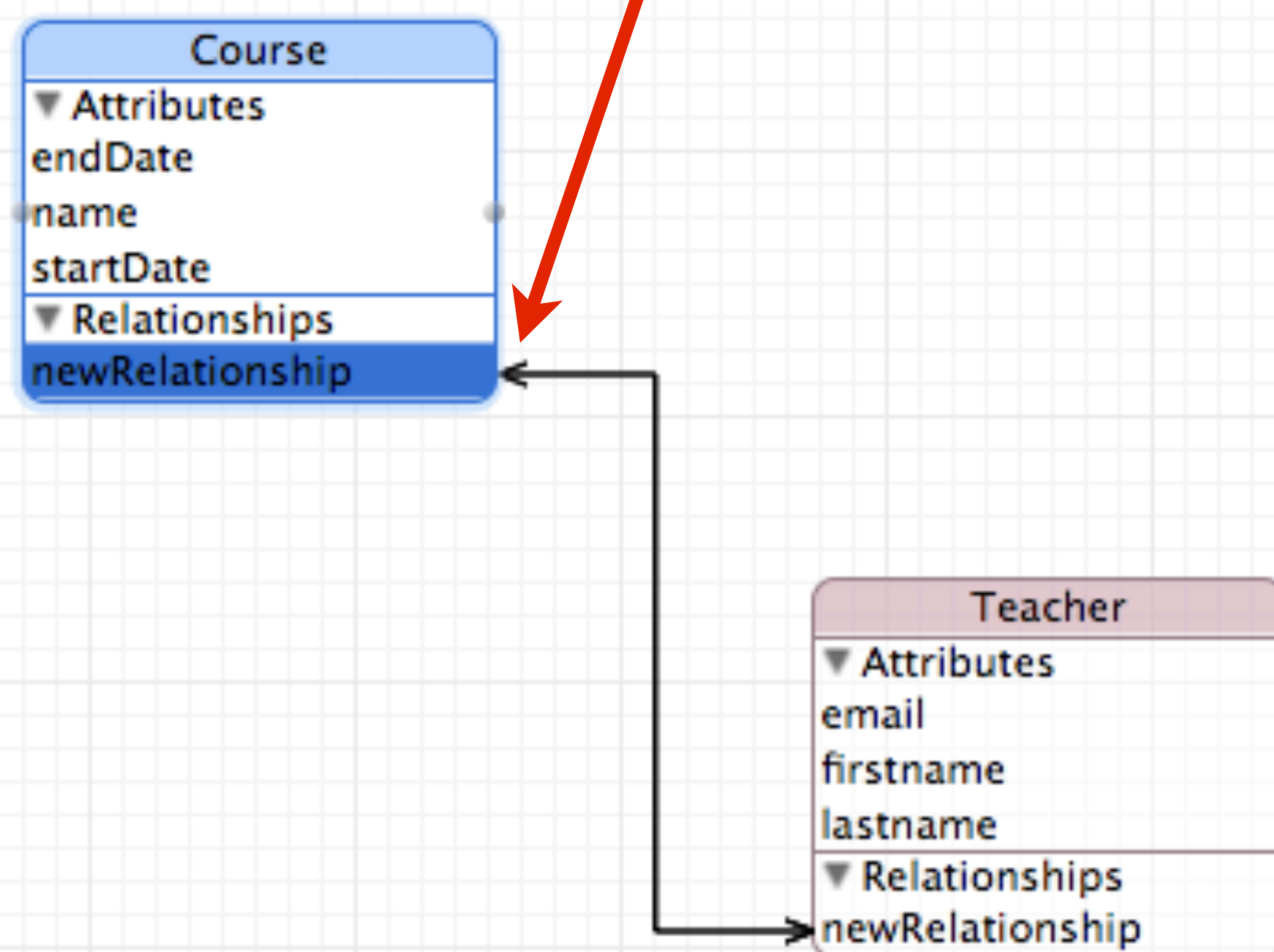
Teacher
Attributes
email
firstname
lastname
Relationships

Outline Style
Add Entity
Add Attribute
Editor Style

This button switches the model view from table to E-R model-like

Setting Relationship properties

Select the relationship to show the Data Model Inspector



Relationship
Name newRelationship
Properties Transient Optional
Destination Teacher
Inverse newRelationship
Delete Rule Nullify
Type To One
Advanced Index in Spotlight
Store in External Record File

User Info
Key Value
+ −

Versioning
Hash Modifier Version Hash Modifier
Renaming ID Renaming Identifier

Setting Relationship properties

Relationship
Name newRelationship
Properties Transient Optional
Destination Teacher
Inverse newRelationship
Delete Rule Nullify
Type To One
Advanced Index in Spotlight
Store in External Record File
User Info
Key Value
Versioning
Hash Modifier Version Hash Modifier
Renaming ID Renaming Identifier

Course
Attributes
endDate
name
startDate
Relationships
newRelationship

Teacher
Attributes
email
firstname
lastname
Relationships
newRelationship

Relationship's name

Setting Relationship properties

Relationship

Name newRelationship

Properties Transient Optional

Destination Teacher

Inverse newRelationship

Delete Rule Nullify

Type To One

Advanced Index in Spotlight

Store in External Record File

User Info

Key Value

Versioning

Hash Modifier Version Hash Modifier

Renaming ID Renaming Identifier

Course
- Attributes
  - endDate
  - name
  - startDate
- Relationships
  - newRelationship

Teacher
- Attributes
  - email
  - firstname
  - lastname
- Relationships
  - newRelationship

Behavior on deletion (when the pointed object gets deleted on the graph)

- No Action
- ✓ Nullify
- Cascade
- Deny

Setting Relationship properties

Relationship

Name: newRelationship
Properties: Transient, Optional (checked)
Destination: Teacher
Inverse: newRelationship
Delete Rule: Nullify
Type: To One
Advanced: Index in Spotlight, Store in External Record File

User Info
Key | Value

Versioning
Hash Modifier: Version Hash Modifier
Renaming ID: Renaming Identifier

Course: Attributes (endDate, name, startDate); Relationships (newRelationship)

Teacher: Attributes (email, firstname, lastname); Relationships (newRelationship)

Cardinality

To Many
✓ To One

Setting Relationship properties

Course
- Attributes
  - endDate
  - name
  - startDate
- Relationships
  - teacher

Teacher
- Attributes
  - email
  - firstname
  - lastname
- Relationships
  - teaches

Relationship

Name	teaches
Properties	☐ Transient ☑ Optional
Destination	Course
Inverse	teacher
Delete Rule	Nullify
Type	To Many
Arrangement	☐ Ordered
Count	Unbounded ☐ Minimum
	Unbounded ☐ Maximum
Advanced	☐ Index in Spotlight
	☐ Store in External Record File

User Info

Key	Value

Versioning

Hash Modifier	Version Hash Modifier
Renaming ID	Renaming Identifier

Inverse relationship

Setting Relationship properties

Course
Attributes
endDate
name
startDate
Relationships
teacher

Teacher
Attributes
email
firstname
lastname
Relationships
teaches

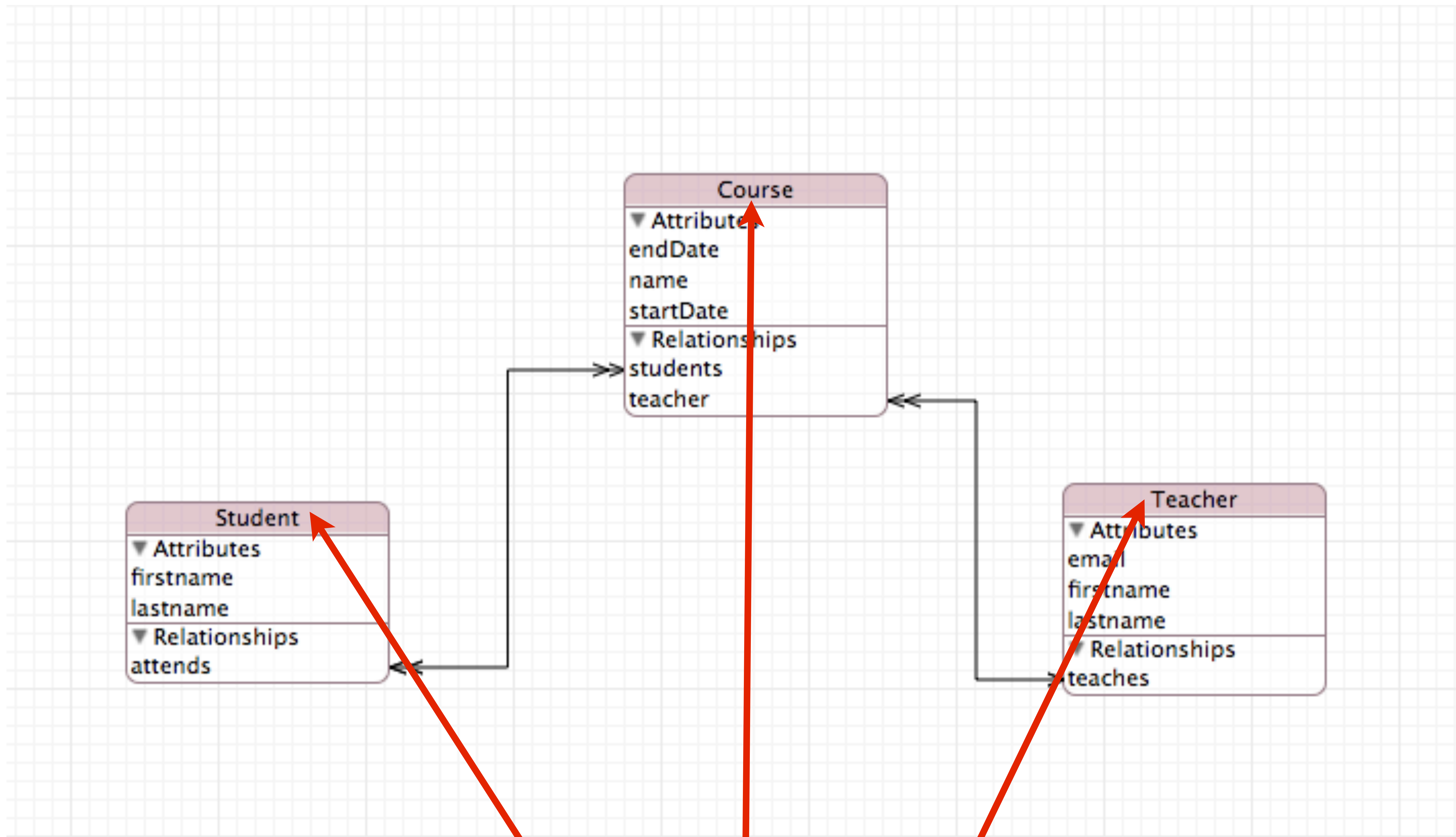
Relationship
Name: teaches
Properties: Transient, Optional
Destination: Course
Inverse: teacher
Delete Rule: Nullify
Type: To Many
Arrangement: Ordered
Count: Unbounded, Minimum
Unbounded, Maximum
Advanced: Index in Spotlight
Store in External Record File
User Info
Key | Value
Versioning
Hash Modifier: Version Hash Modifier
Renaming ID: Renaming Identifier

Setting “To Many” for the cardinality changes the type of arrow in the view

Setting Relationship properties

It is possible to set the range for the cardinality and whether the pointed objects should be ordered

Setting Relationship properties



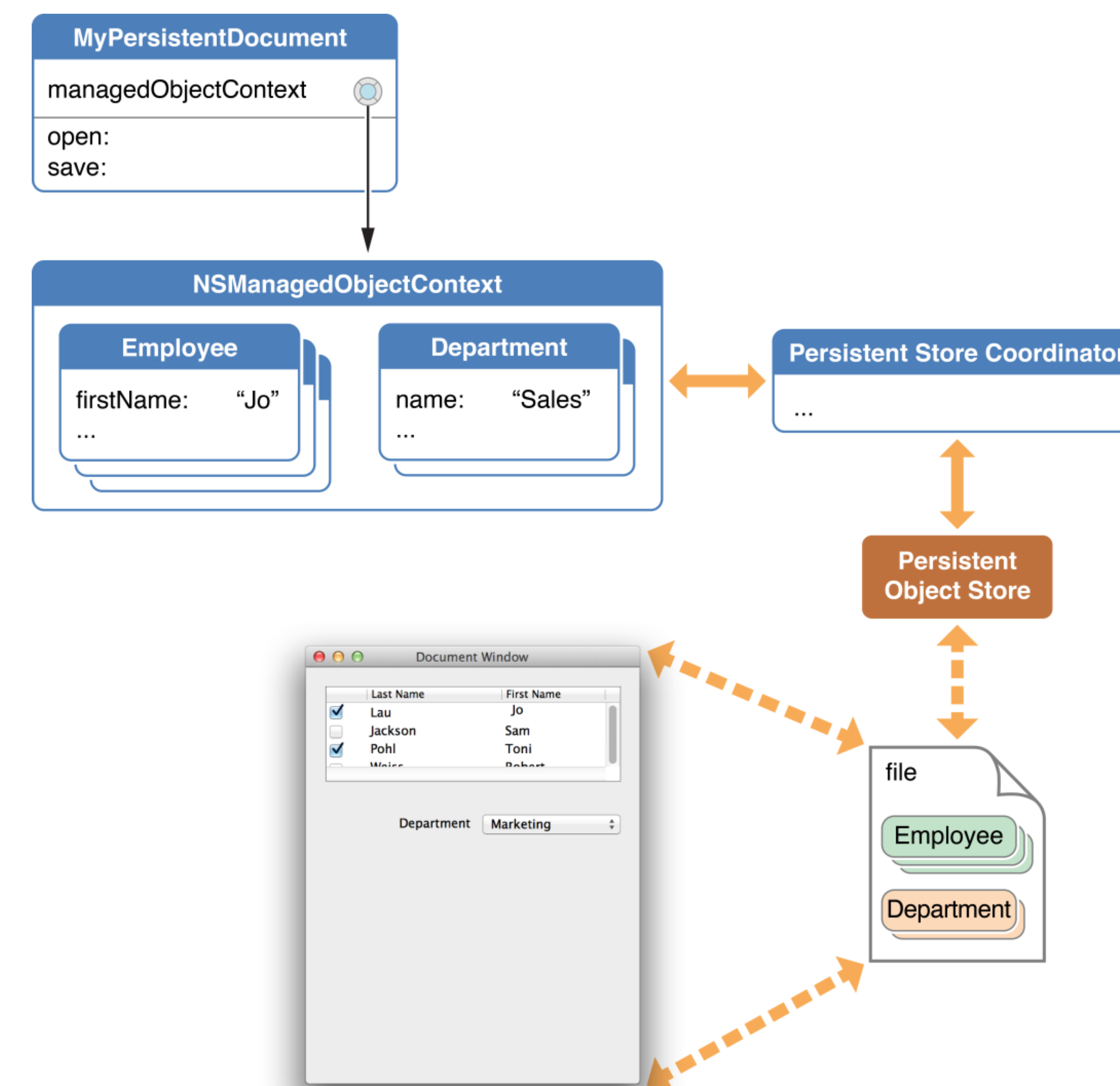
Entities in code will be `NSManagedObject` or subclass of `NSManagedObject`

Using the Data Model in code

- The Data Model contains more information than just entities, attributes, and relationships, such as the layout of the diagram, its colors of elements, and so on, which are not needed at runtime
- The Data Model file is compiled using the model compiler, **momc**, to remove the excess information and make runtime loading of the resource efficient
- An **xcdatamodeld** "source" directory is compiled into a **momd** deployment directory, and an **xcdatamodel** "source" file is compiled into a **mom** deployment file
- The Model (an instance of `NSManagedObjectModel`) can be loaded in many ways:
  - if using a Document-based application template, no code is needed to load the model
  - otherwise, a single model can be loaded from a specific URL using the `initWithContentsOfURL:` initializer (*preferred technique*)
  - finally, it can be created by merging many models using the mergedModelFromBundles: method

Managed object context

- Core Data functionalities are supported in code through an `NSManagedObjectContext` (or just **context**)
- The managed object context serves as as gateway to an underlying collection of framework objects, collectively known as the persistence stack, that mediate between the objects in the application and external data stores
- Objects managed by the context are called **managed objects**
- When objects are fetched from a persistent store, temporary copies are loaded to form an object graph (or a collection of object graphs)
- Managed objects can be modified and will then replace the ones in the persistent store when they are saved



Managed object context

- All managed objects must be registered with a managed object context
- Objects can be added to and removed from the graph using the context
- The context keeps track of the changes made both to individual objects' attributes and to the relationships between objects
- By tracking changes, the context is able to provide undo and redo support
- The context also ensures that if a relationship between objects is changed, the integrity of the object graph is maintained
- There may be more than one managed object context in an application
- A given object in a persistent store may be edited in more than one context simultaneously: be careful to avoid inconsistencies when performing save operations!

Getting a managed object context

- A managed object context can be obtained in two ways:
    - the “Use Core Data” option can be checked (only with specific project templates) when creating a new project to have the application delegate include a `managedObjectContext` property
    - create a UIManagedDocument and get the managed object context with its `managedObjectContext` property

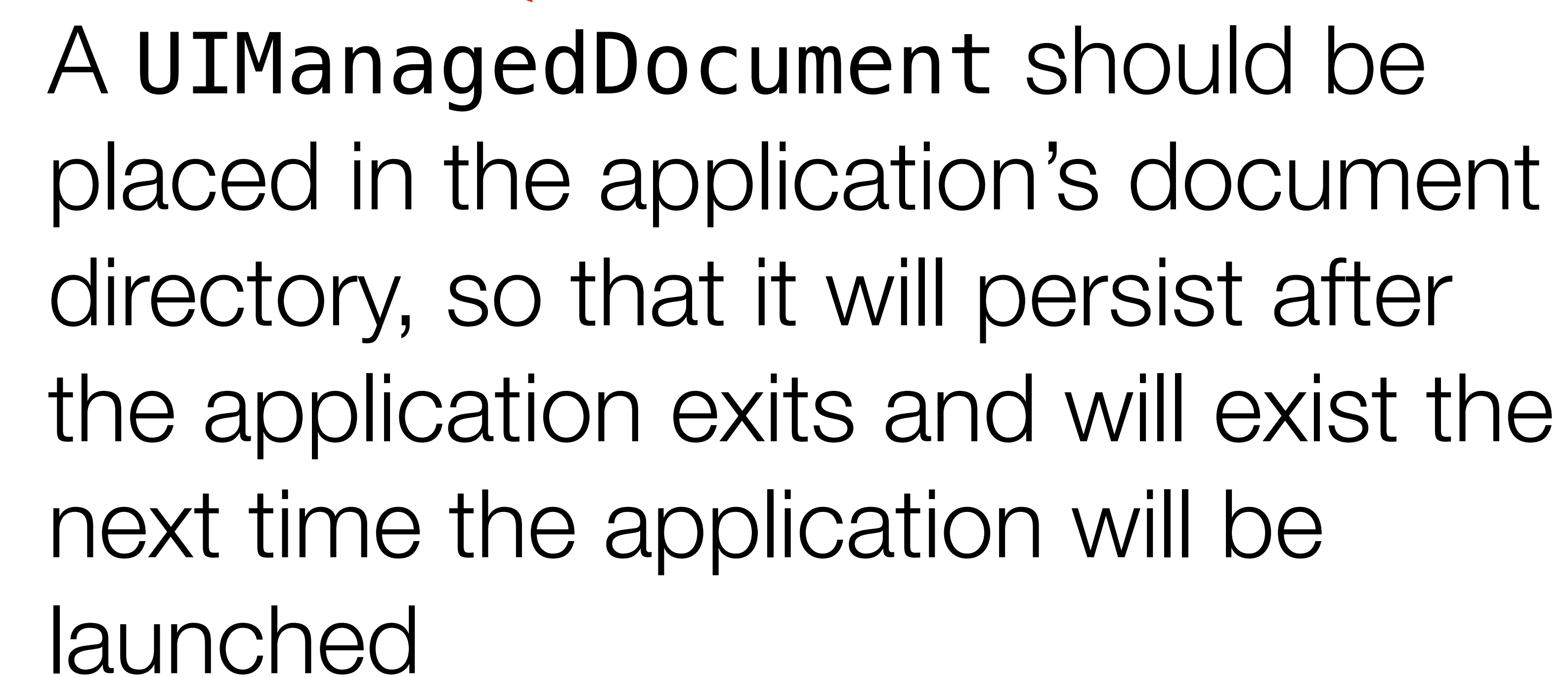
UIManagedDocument

- UIKit defines the class `UIManagedDocument`, a concrete subclass of `UIDocument`
- `UIDocument` is an abstract class that provides mechanisms for:
  - asynchronous reading and writing of data on a background queue
  - integration with cloud services (iCloud)
  - discovering of conflicts between different versions of a document
  - safe-saving of document data by writing data first to a temporary file and then replacing the current document file with it
  - automatic saving of document data at opportune moments
- `UIManagedDocument` is a subclass of `UIDocument` which integrates with Core Data
- A managed document is initialized by specifying the URL for the document location

UIManagedDocument

1) Create a `UIManagedDocument`

```
NSFileManager *fileManager = [NSFileManager defaultManager];
NSURL *documentsDirectory = [[fileManager URLsForDirectory:NSDocumentationDirectory
inDomains:NSUserDomainMask] firstObject];
NSURL *url = [documentsDirectory URLByAppendingPathComponent:@"Document"];
UIManagedDocument *document = [[UIManagedDocument alloc] initWithFileURL:url];
BOOL fileExists = [fileManager fileExistsAtPath:[url path]];
if(fileExists == YES){
    [document openWithCompletionHandler:^(BOOL success) {
        /* on open */
    }];
}
else{
    [document saveToURL:url
       forSaveOperation:UIDocumentSaveForCreating
      completionHandler:^(BOOL success) {
        /* on create */
    }];
}
```

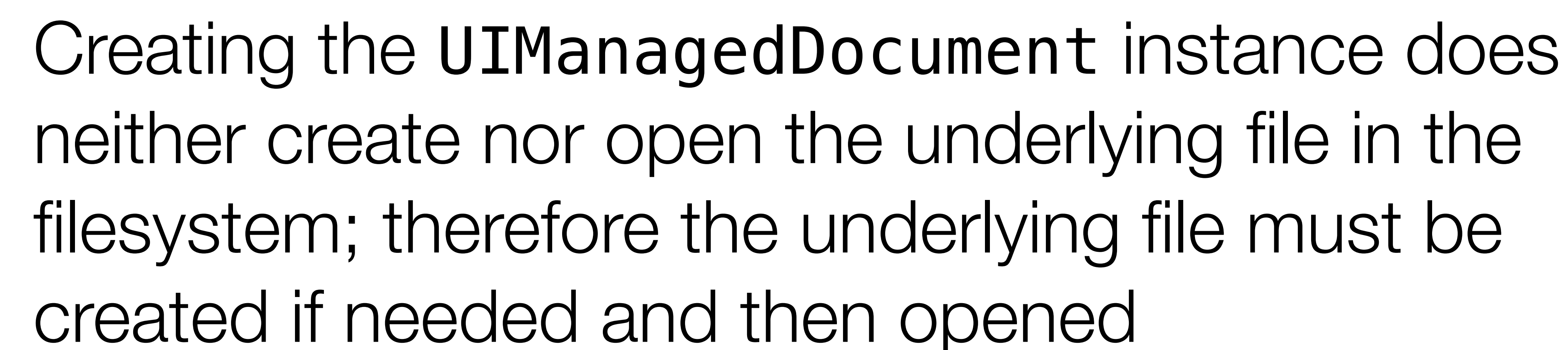


A `UIManagedDocument` should be placed in the application's document directory, so that it will persist after the application exits and will exist the next time the application will be launched

UIManagedDocument

1) Create a `UIManagedDocument`

```
NSFileManager *fileManager = [NSFileManager defaultManager];
NSURL *documentsDirectory = [[fileManager URLsForDirectory:NSDocumentationDirectory
inDomains:NSUserDomainMask] firstObject];
NSURL *url = [documentsDirectory URLByAppendingPathComponent:@"Document"];
UIManagedDocument *document = [[UIManagedDocument alloc] initWithFileURL:url];
BOOL fileExists = [fileManager fileExistsAtPath:[url path]];
if(fileExists == YES){
    [document openWithCompletionHandler:^(BOOL success) {
        /* on open */
    }];
}
else{
    [document saveToURL:url
       forSaveOperation:UIDocumentSaveForCreating
      completionHandler:^(BOOL success) {
        /* on create */
    }];
}
```



UIManagedDocument

2) Check for the existence of the underlying file

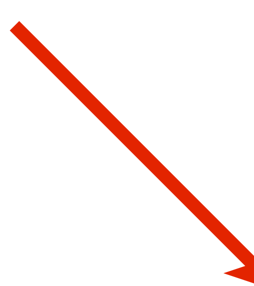
```
NSFileManager *fileManager = [NSFileManager defaultManager];
NSURL *documentsDirectory = [[fileManager URLsForDirectory:NSDocumentationDirectory
 inDomains:NSUserDomainMask] firstObject];
NSURL *url = [documentsDirectory URLByAppendingPathComponent:@"Document"];
UIManagedDocument *document = [[UIManagedDocument alloc] initWithFileURL:url];
BOOL fileExists = [fileManager fileExistsAtPath:[url path]];
if(fileExists == YES){
    [document openWithCompletionHandler:^(BOOL success) {
        /* on open */
    }];
}
else{
    [document saveToURL:url
       forSaveOperation:UIDocumentSaveForCreating
      completionHandler:^(BOOL success) {
        /* on create */
    }];
}
```

Creating the **`UIManagedDocument`** instance does neither create nor open the underlying file in the filesystem; therefore the underlying file must be created if needed and then opened

UIManagedDocument

3) Open the file if it exists

```
NSFileManager *fileManager = [NSFileManager defaultManager];
NSURL *documentsDirectory = [[fileManager URLsForDirectory:NSDocumentationDirectory
inDomains:NSUserDomainMask] firstObject];
NSURL *url = [documentsDirectory URLByAppendingPathComponent:@"Document"];
UIManagedDocument *document = [[UIManagedDocument alloc] initWithFileURL:url];
BOOL fileExists = [fileManager fileExistsAtPath:[url path]];
if(fileExists == YES){
    [document openWithCompletionHandler:^(BOOL success) {
        /* on open */
    }];
}
else{
    [document saveToURL:url
       forSaveOperation:UIDocumentSaveForCreating
      completionHandler:^(BOOL success) {
        /* on create */
    }];
}
```



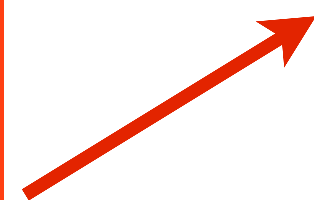
If the file exists, it can be opened

UIManagedDocument

4) Create the file if it doesn't exist

```
NSFileManager *fileManager = [NSFileManager defaultManager];
NSURL *documentsDirectory = [[fileManager URLsForDirectory:NSDocumentationDirectory
inDomains:NSUserDomainMask] firstObject];
NSURL *url = [documentsDirectory URLByAppendingPathComponent:@"Document"];
UIManagedDocument *document = [[UIManagedDocument alloc] initWithFileURL:url];
BOOL fileExists = [fileManager fileExistsAtPath:[url path]];
if(fileExists == YES){
    [document openWithCompletionHandler:^(BOOL success) {
        /* on open */
    }];
}
else{
    [document saveToURL:url
       forSaveOperation:UIDocumentSaveForCreating
      completionHandler:^(BOOL success) {
        /* on create */
    }];
}
```

If the file doesn't exist, it must be created



UIManagedDocument

The open/create operations are asynchronous and occur on a separate queue

```
if(fileExists == YES){
    [document openWithCompletionHandler:^(BOOL success) {
        /* on open */
    }];
}
else{
    [document saveToURL:url
       forSaveOperation:UIDocumentSaveForCreating
      completionHandler:^(BOOL success) {
        /* on create */
    }];
}
```

UIManagedDocument

```
if(fileExists == YES){
    [document openWithCompletionHandler:^(BOOL success) {
        /* on open */
    }];
}
else{
    [document saveToURL:url
       forSaveOperation:UIDocumentSaveForCreating
      completionHandler:^(BOOL success) {
        /* on create */
    }];
}
```

When the open/create operations are completed the completion handler blocks are executed

UIManagedDocument

```
if(fileExists == YES){
    [document openWithCompletionHandler:^(BOOL success) {
        /* on open */
    }];
}
else{
    [document saveToURL:url
       forSaveOperation:UIDocumentSaveForCreating
      completionHandler:^(BOOL success) {
        /* on create */
    }];
}
```

The `success` argument tells whether the operation completed successfully or failed

UIManagedDocument example

```
- (void)createDocument{
  NSFileManager *fileManager = [NSFileManager defaultManager];
  NSURL *documentsDirectory = [[fileManager URLsForDirectory:NSDocumentationDirectory
                                                   inDomains:NSUserDomainMask] firstObject];
  NSURL *url = [documentsDirectory URLByAppendingPathComponent:@"Document"];
  UIManagedDocument *document = [[UIManagedDocument alloc] initWithFileURL:url];
  BOOL fileExists = [fileManager fileExistsAtPath:[url path]];
  if(fileExists == YES){
      [document openWithCompletionHandler:^(BOOL success) {
         /* on open */
         if(success == YES){
            [self onDocumentReady];
         }
         else{
            /* handle error */
         }
      }];
  }
  else{
      [document saveToURL:url
         forSaveOperation:UIDocumentSaveForCreating
        completionHandler:^(BOOL success) {
           /* on create */
           if(success == YES){
              [self onDocumentReady];
           }
           else{
              /* handle error */
           }
        }];
  }
}
```

Before using the document, check for the document state:

```
UIDocumentStateNormal
UIDocumentStateClosed
UIDocumentStateEditingDisabled
UIDocumentStateInConflict
UIDocumentStateSavingError
```

```
- (void)onDocumentReady{
  if(self.document.documentState == UIDocumentStateNormal){
     /* use document here */
  }
}
```

Saving UIManagedDocument

- `UIManagedDocument` instances autosave so do not need to be explicitly saved
- In case they needed to be saved manually:

```
[self.document saveToURL:self.document.fileURL
       forSaveOperation:UIDocumentSaveForOverwriting
      completionHandler:^(BOOL success) {

            /* on save: save operation is asynchronous! */

}];
```

- `UIManagedDocument` instances must operate on the main queue

Closing UIManagedDocument

- `UIManagedDocument` instances close automatically when there is no strong pointer pointing to it
- In case they needed to be closed manually:

```
[self.document closeWithCompletionHandler:^(BOOL success) {
    /* on close: close operation is asynchronous! */
}];
```

Observing NSManagedObjectContext

- `NSManagedObjectContext` posts notification whenever anything significant happens
- The notification object is the `NSManagedObjectContext` instance
- The `userInfo` dictionary of the notification object is a dictionary with the following keys:
  - `NSInsertedObjectsKey`: array of inserted `NSManagedObject` instances
  - `NSUpdatedObjectsKey`: array of updated `NSManagedObject` instances
  - `NSDeletedObjectsKey`: array of deleted `NSManagedObject` instances
- Notifications posted:
  - `NSManagedObjectContextObjectsDidChangeNotification`: values of properties of objects contained in a managed object context are changed
  - `NSManagedObjectContextDidSaveNotification`: a managed object context completes a save operation
  - `NSManagedObjectContextWillSaveNotification`: a managed object context is about to perform a save operation (no `userInfo` dictionary is not set)

Inserting objects

- Objects are inserted into the database with the following method:

```
NSManagedObjectContext *context = self.document.managedObjectContext;
NSManagedObject *teacher = [NSEntityDescription insertNewObjectForEntityForName:@"Teacher"
                                                        inManagedObjectContext:context];
```

- An `NSEntityDescription` specifies the name of an entity, the class used to represent the entity, and the entity's properties
- The objects returned are `NSManagedObject` instances
- `NSManagedObject` instances are the counterparts in code of the entity defined in the Data Model
- The returned `NSManagedObject` instances' attributes are `nil`, unless a default value has been set
- The `NSManagedObject` class supports Key-Value Coding; this means that it is possible to access the attributes of a managed object the following methods can be used:

```
- (void)setValue:(id)value forKey:(NSString *)key
- (id)valueForKey:(NSString *)key
```

- The `key` is the attribute name and `value` is the data that will be stored

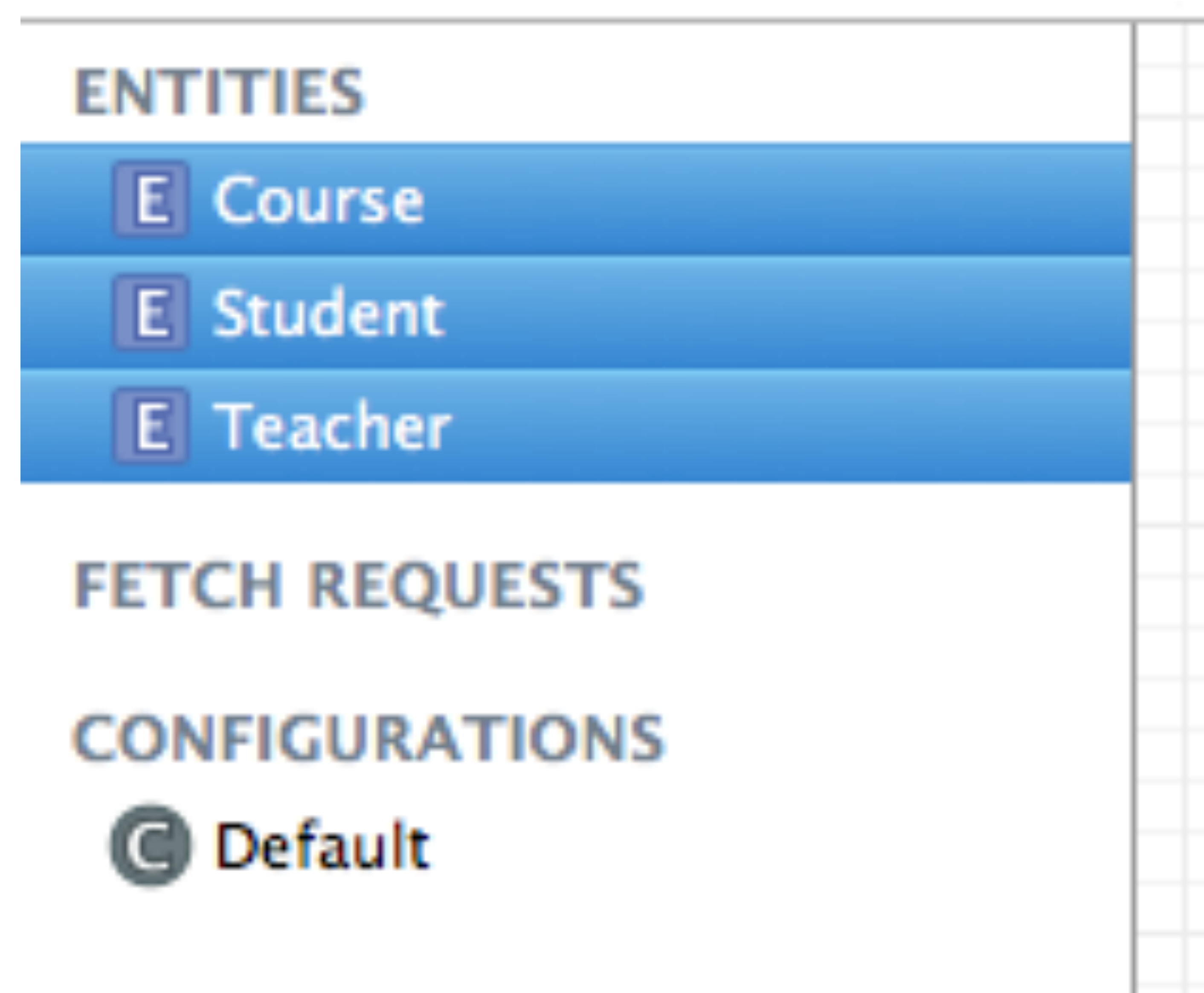
```
[teacher setValue:@"Simone" forKey:@"firstname"];
[teacher setValue:@"Cirani" forKey:@"lastname"];
[teacher setValue:@"simone.cirani@unipr.it" forKey:@"email"];
```

Using properties to access attributes

- Using Key-Value Coding introduces some problems into the code:
  - no type checking can be performed (on the values)
  - the code is filled with literals (on the keys)
- The best way to work with attributes would be to use properties, just like if entities were regular Objective-C classes
- To do so, appropriate subclasses of `NSManagedObject` can be created from Xcode
- `NSManagedObject` subclasses have the same name of their corresponding entity
- Each `NSManagedObject` subclass will have a property for each of its attributes
- Xcode will generate the header and implementation files automatically

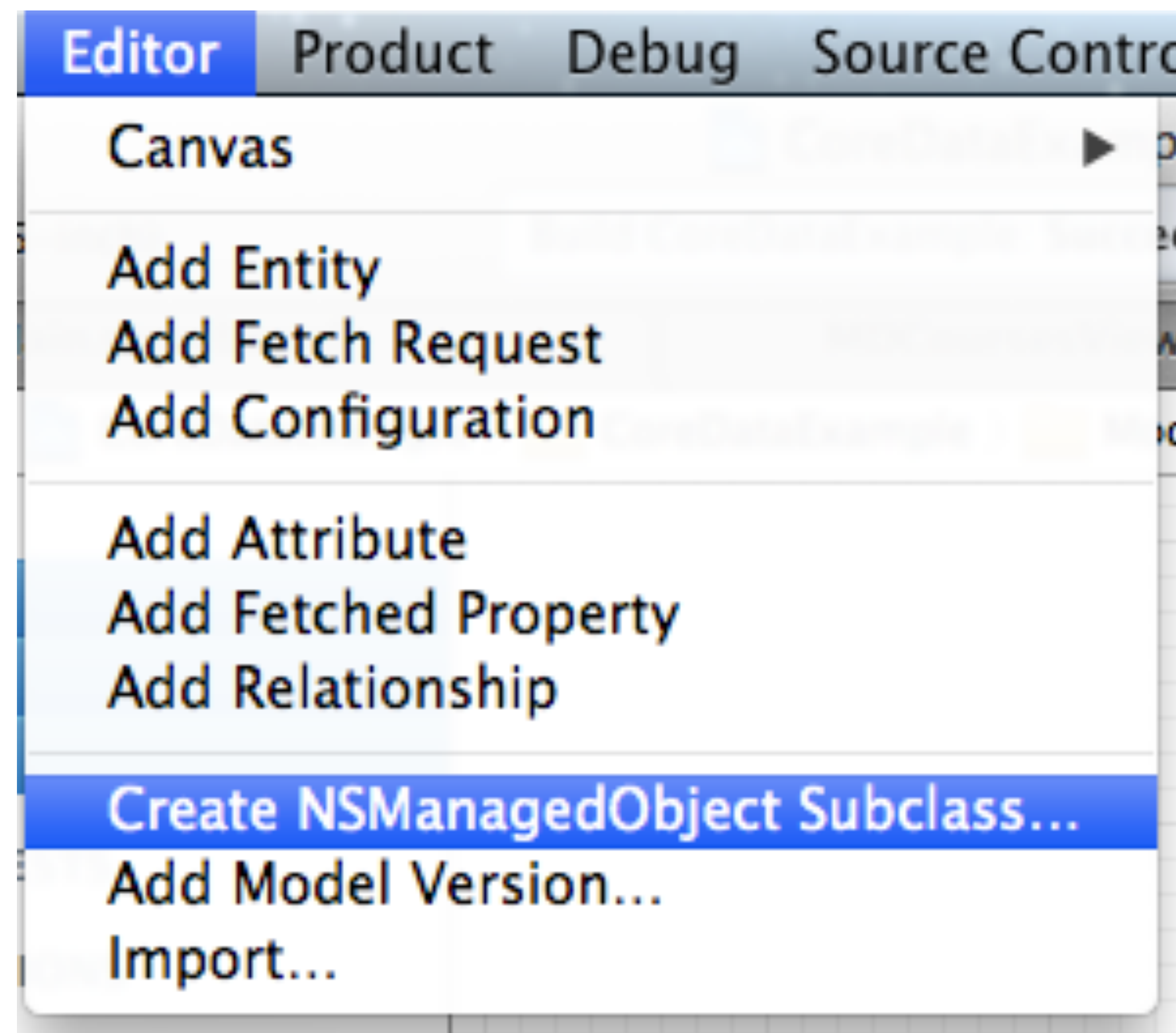
Create subclasses of NSManagedObject

1) Select all the entities that must have a corresponding Objective-C NSManagedObject subclass



Create subclasses of NSManagedObject

2) In the Editor menu, select “Create NSManagedObject subclass”



Create subclasses of NSManagedObject

3) Select which entities Xcode will generate code for

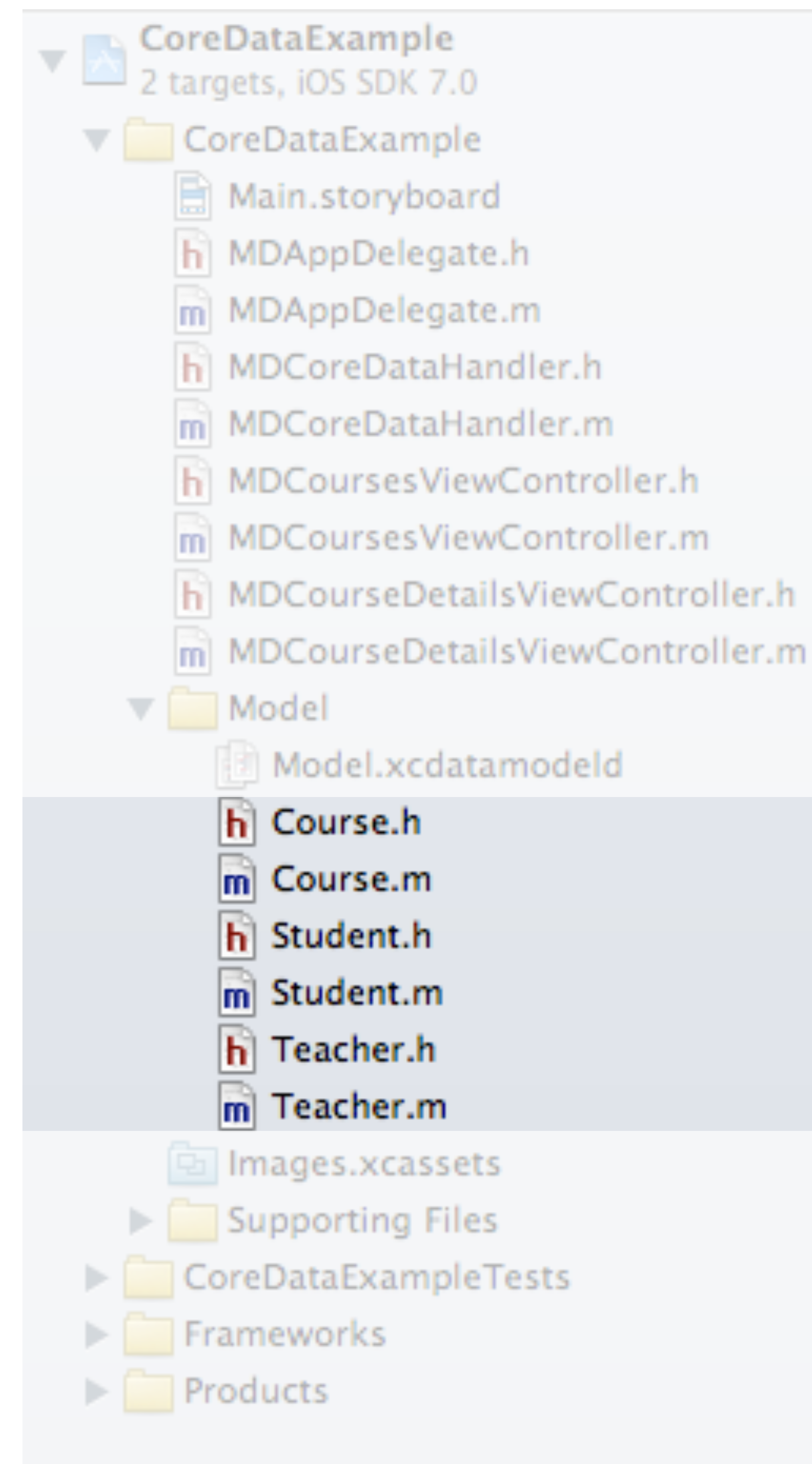
Select the entities you would like to manage

Select	Entity
☑	Student
☑	Course
☑	Teacher

Cancel | Previous | Next

Create subclasses of NSManagedObject

4) In the Navigator, the new classes appear



Create subclasses of NSManagedObject

5) The interface file defines all the properties for the attributes and the relationships

```
#import <Foundation/Foundation.h>
#import <CoreData/CoreData.h>


@interface Teacher : NSManagedObject

@property (nonatomic, retain) NSString * firstname;
@property (nonatomic, retain) NSString * lastname;
@property (nonatomic, retain) NSString * email;
@property (nonatomic, retain) NSSet *teaches;
@end

@interface Teacher (CoreDataGeneratedAccessors)

- (void)addTeachesObject:(NSManagedObject *)value;
- (void)removeTeachesObject:(NSManagedObject *)value;
- (void)addTeaches:(NSSet *)values;
- (void)removeTeaches:(NSSet *)values;

@end
```

Create subclasses of NSManagedObject

6) The implementation file does not synthesize properties, but uses the `@dynamic` directive; the implementation file could be empty, the properties are marked as `@dynamic` just to suppress warnings

```
#import "Teacher.h"


@implementation Teacher

@dynamic firstname;
@dynamic lastname;
@dynamic email;
@dynamic teaches;

@end
```

`@dynamic` means that the class does not implement the setter and getter for the properties, but it will use a “trap” mechanism to let the runtime avoid throwing an exception (because the method is not implemented) and just call `valueForKey:` and `setValue:ForKey:` instead

Using properties to access attributes

- At this point, it is possible to use a specific Objective-C class for the entity and its properties to access the attributes:

```
NSManagedObjectContext *context = self.document.managedObjectContext;

Teacher *teacher = [NSEntityDescription insertNewObjectForEntityForName:@"Teacher"
                                                 inManagedObjectContext:context];
teacher.firstname = @"Simone";
teacher.lastname = @"Cirani";
teacher.email = @"simone.cirani@unipr.it”;
```

Adding methods and properties to NSManagedObject subclasses

- Sometimes, the default implementation generated by Xcode are not sufficient for the purposes of the application
- It could be necessary to have other methods and properties (or custom accessor methods), for instance a sort of initializer that inserts a new object into the database and returns it
- Adding methods and properties directly to the generated files is dangerous: if the model gets modified and they must be regenerated, all the changes would be lost
- To avoid this risk, categories can be used to add behavior to the generated classes
- This way the new methods and properties won't be affected by changes in the model

Adding methods and properties to NSManagedObject subclasses

```
#import "Teacher.h"

@interface Teacher (Create)

+ (Teacher *)createWithDictionary:(NSDictionary *)dictionary
           inManagedObjectContext:(NSManagedObjectContext *)context;

@end
```

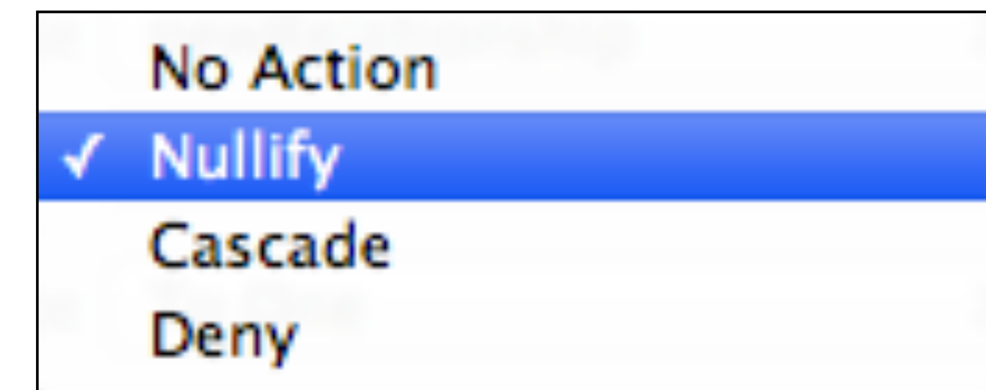
```
#import "Teacher+Create.h"

@implementation Teacher (Create)

+ (Teacher *)createWithDictionary:(NSDictionary *)dictionary
           inManagedObjectContext:(NSManagedObjectContext *)context{
    Teacher *teacher = [NSEntityDescription insertNewObjectForEntityForName:@"Teacher"
                                                     inManagedObjectContext:context];

    /* initialize the object and perform other needed operations */

    return teacher;
}
```

Deleting objects

- Objects are deleted with the following method:

```
NSManagedObjectContext *context = self.document.managedObjectContext;
[context deleteObject:teacher];
```

- This removes the managed object from the object graph
- Just as a new object is not saved to the store until the context is saved, a deleted object is not removed from the store until the context is saved
- When an object with relationships is deleted, what happens to the destination object of the relationship? It depends on the Delete rule defined when setting the relationship's properties:
  - **Nullify**: set the inverse relationship for objects at the destination to null
  - **Cascade**: delete the objects at the destination of the relationship
  - **Deny**: if there is at least one object at the relationship destination, then the source object cannot be deleted
  - **No Action**: do nothing to the object at the destination of the relationship



Deleting objects

- It is possible to take some action right before the deletion is performed
- The `prepareForDeletion` method can be implemented to do perform other operations that are needed to preserve the managed object's integrity
- For example, a `Teacher` object might have a `numberOfCourses` properties; if a `Course` is deleted, then the value of `numberOfCourses` should be decreased by one

```
- (void)prepareForDeletion{
    self.teacher.numberOfCourses = @([self.teacher.numberOfCourses intValue] - 1);
}
```

- The `prepareForDeletion` method should be placed in a category as well

Fetching objects with NSFetchRequest

- Fetching (or querying for) objects in the database is performed by executing an `NSFetchRequest` in the `NSManagedObjectContext`
- An `NSFetchRequest` must be configured in order to define:
  - which entity to fetch
  - (optionally) how many objects to be fetched (a sort of pagination)
  - the sorting of the fetched objects (an array of `NSSortDescriptor` objects)
  - an `NSPredicate`, which is basically the clause that specifies which objects should be fetched

NSSortDescriptor

- An `NSFetchRequest` returns an `NSArray` of `NSManagedObject` instances
- An `NSSortDescriptor` object defines the order that the objects in the returned array should have
- More than one sort descriptor can be specified for complex ordering (first order by last name, then by first name)
- An `NSSortDescriptor` is created in the following way

```
NSSortDescriptor *sortDescriptor =
                [NSSortDescriptor sortDescriptorWithKey:@"lastname"
                                              ascending:YES
                                               selector:@selector(localizedStandardCompare:)];
```

- The `key` is the attribute to use for ordering
- The `ascending` argument specifying the direction of the ordering (SQL's ASC or DESC)
- The `selector` is a method used to compare objects and decide whether an object should be before or after another in the order
- `localizedStandardCompare:` is a standard convenience method for string ordering

NSSortDescriptor

- An `NSFetchRequest` returns an `NSArray` of `NSManagedObject` instances
- When an `NSSortDescriptor` has been created, it can be used to configure An `NSFetchRequest`:

```
request.sortDescriptors = @[sortDescriptor];
```

- If many sort descriptors have been defined:

```
request.sortDescriptors = @[sortDescriptor1, sortDescriptor2];
```

NSPredicate

- The `NSPredicate` class is used to define logical conditions used to constrain a search either for a fetch
- Predicates are often created from a format string which is parsed by the class methods on `NSPredicate`
- The `NSPredicate` is created from an `NSString`, but the content have a specific (semantic) meaning:

```
NSString *lastname = ...;
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"lastname = %@",lastname];
```

- Typical predicates are:
  - simple comparisons, such as `firstname = "Simone"` or `firstname like "Simo"`
  - case insensitive lookups, such as `name contains[c] "mobile"`
  - logical operations, such as `(firstname like "Simone") OR (lastname like "Cirani")`
- It is possible to create predicates also for relationships:
  - `course.teacher.lastname = "Cirani"`
- Refer to https://developer.apple.com/library/ios/documentation/cocoa/conceptual/Predicates/predicates.html for more details on predicate syntax

NSCompoundPredicate

- NSCompoundPredicates are used to combine logically (AND/OR/NOT) many predicates
- Multiple NSPredicates are created normally...

```
NSPredicate *pr1 = ...;
NSPredicate *pr2 = ...;
```

- ...then added to an array (of subpredicates)...

```
NSArray *array = @[pr1,pr2];
```

- ...finally the array is used to create a compound predicate...

```
NSPredicate *predicate = [NSCompoundPredicate andPredicateWithSubpredicates:array];
```

- Compound predicates can be created with AND/OR/NOT logical statements with the following methods:
    - `andPredicateWithSubpredicates:`
    - `orPredicateWithSubpredicates:`
    - `notPredicateWithSubpredicate:`

Fetching objects with NSFetchRequest

- An `NSFetchRequest` can therefore be configured as follows:

```
NSFetchRequest *request = [NSFetchRequest fetchRequestWithEntityName:@"Teacher"];

request.fetchBatchSize = 50; // fetch 50 objects at a time

request.fetchLimit = 200; // stop fetching after the first 200 objects

NSSortDescriptor *sortDescriptor =
                [NSSortDescriptor sortDescriptorWithKey:@"lastname"
                                              ascending:YES
                                               selector:@selector(localizedStandardCompare:)];
request.sortDescriptors = @[sortDescriptor];

NSString *lastname = ...;
NSPredicate *predicate = [NSPredicate predicateWithFormat:@"lastname = %@",lastname];

request.predicate = predicate;
```

Fetching objects with NSFetchRequest

- A properly configured `NSFetchRequest` can be executed in the following way:

```
NSManagedObjectContext *context = self.document.managedObjectContext;
NSError *error;
NSArray *teachers = [context executeFetchRequest:request error:&error];
```

- The returned array is `nil` in case an error occurs; in such a case, the `error` variable holds the error

- In case no match is found, the returned array is empty but not `nil`

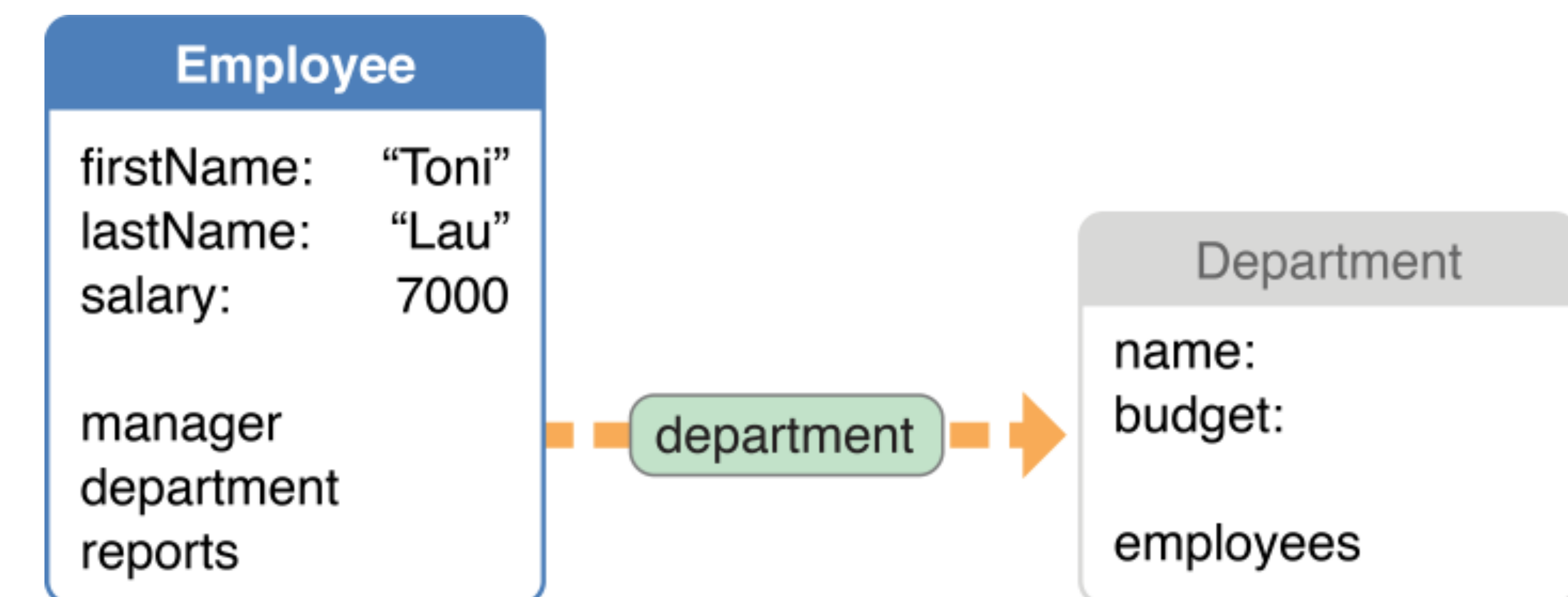
Faulting

- Core Data uses an efficient mechanism to reduce memory usage, called **faulting**
- A **fault** is a placeholder object that represents either:

  a) a managed object that has not yet been fully realized (an instance of the appropriate class, but its persistent variables are not yet initialized)

  b) a collection object that represents a relationship (a relationship fault is a subclass of the collection class that represents the relationship)

- Core Data automatically **fires faults** when necessary (when a persistent property of a fault is accessed)



Faulting

```
NSManagedObjectContext *context = self.document.managedObjectContext;
NSError *error;

NSArray *teachers = [context executeFetchRequest:request error:&error];
```

```
for(Teacher *teacher in teachers){

    NSLog(@"Teacher %@ fetched",teacher.lastname);

}
```

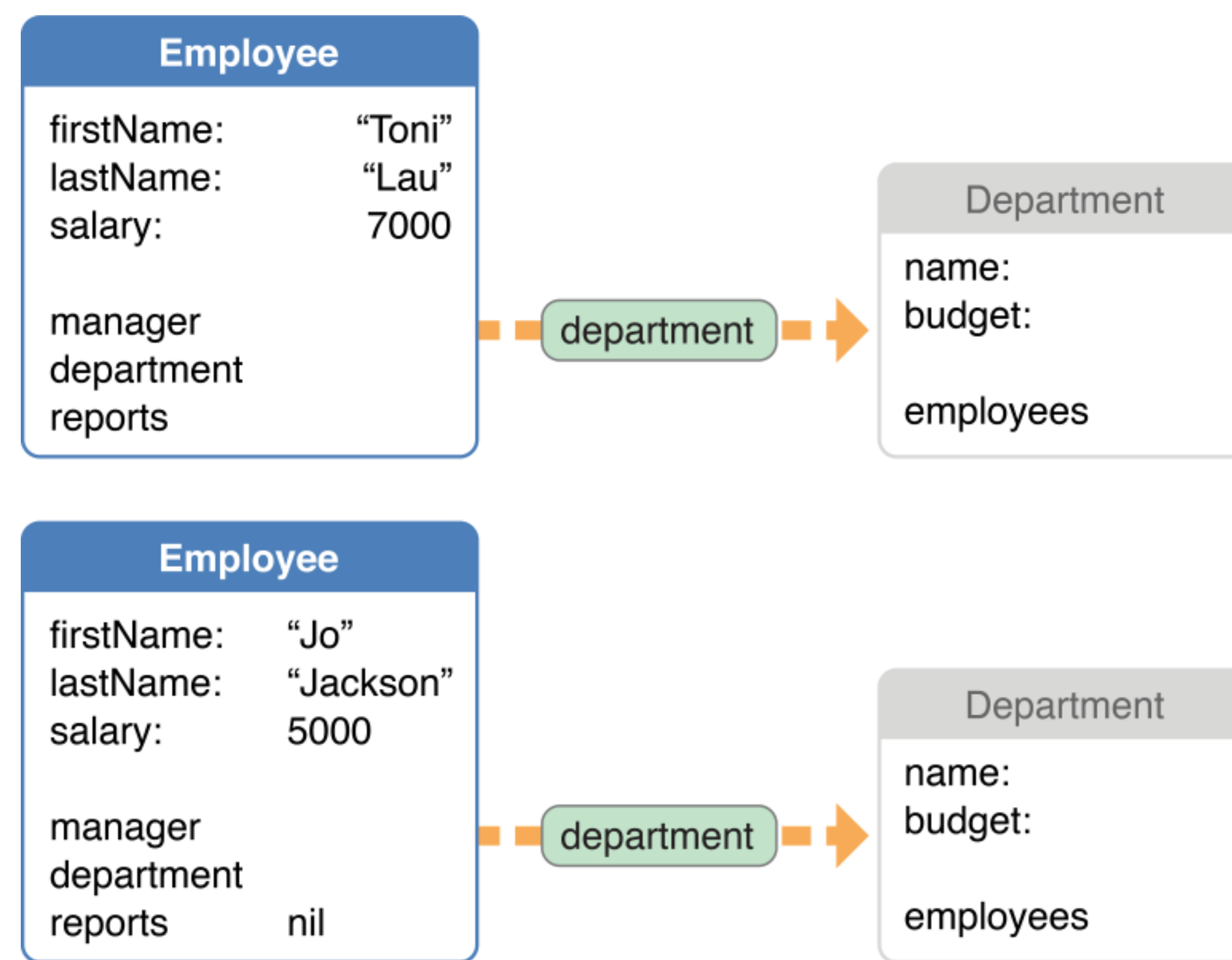
Faulting

```
NSManagedObjectContext *context = self.document.managedObjectContext;
NSError *error;

NSArray *teachers = [context executeFetchRequest:request error:&error];


for(Teacher *teacher in teachers){

    NSLog(@"Teacher %@ fetched",teacher.lastname);

}
```

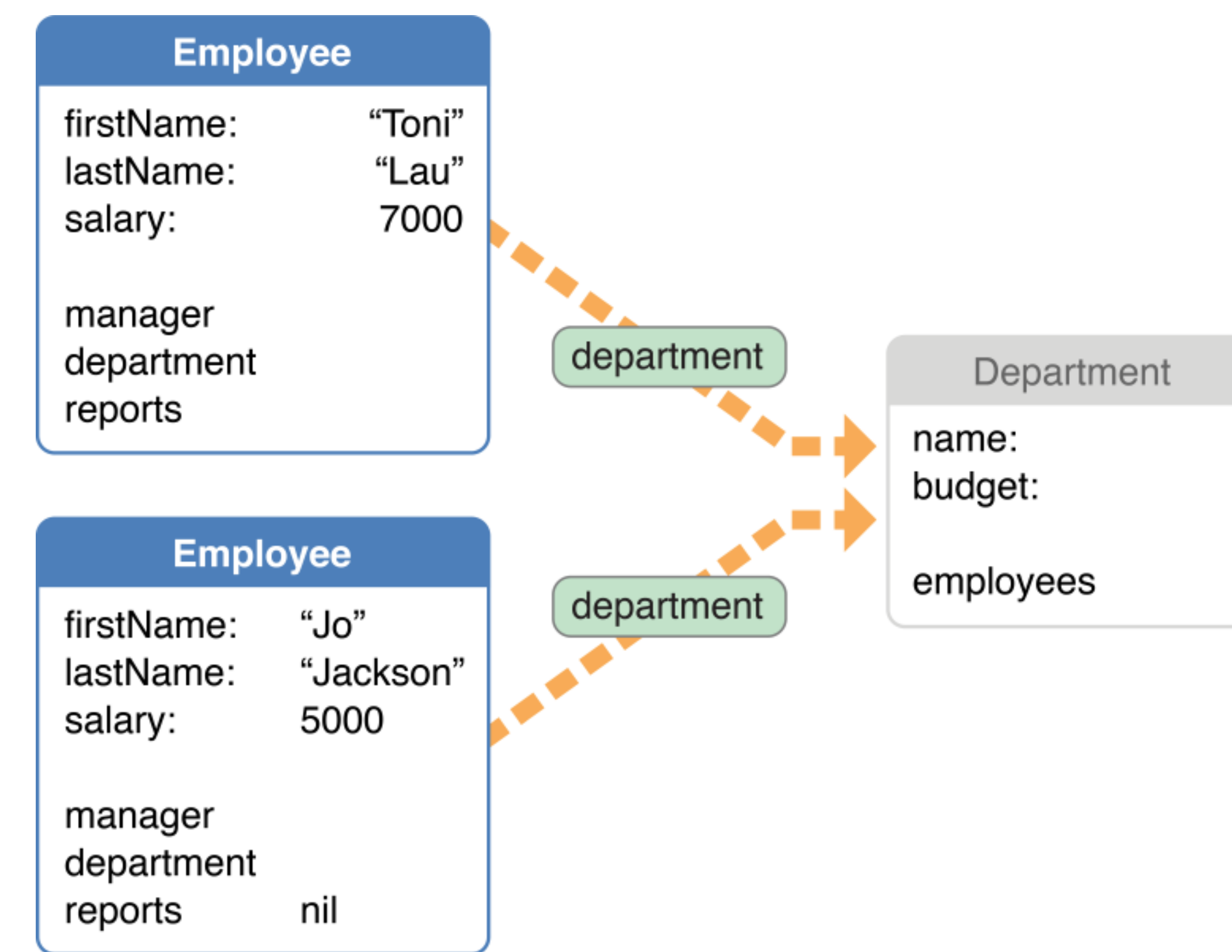
The faults are fired when accessing the property

Uniquing

- **Uniquing** ensures that, in a given managed object context, no more than one managed object to represent a given record exists



without uniquing



with uniquing

- Uniquing avoids that conflicts occur when saving changes to a managed object context, since there is no ambiguity (only one managed object exists)

Concurrency in Core Data

- `NSManagedObjectContext` and the `NSManagedObject` instances in that context are accessible only in the queue they were created
- `NSManagedObjectContext` is not thread-safe
- To access an `NSManagedObjectContext` safely the `performBlock:` and `performBlockAndWait:` methods can be used
- These methods execute the block (asynchronously or synchronously, respectively) in the same queue it was created

Displaying results in table views

- Core Data was designed to work efficiently together with `UITableViews`
- `NSFetchedResultsController` is a class that sets up a bridge between an `NSFetchRequest` and a `UITableViewController`
- `UITableView` expects its data source to provide cells as an array of sections made up of rows
- An `NSFetchedResultsController` can be configured and used to inject objects stored in Core Data inside a `UITableView` in the application's user interface
- `NSFetchedResultsController` also provides mechanisms for caching and monitoring changes to properties for the managed objects

Working with NSFetchedResultsController

- An instance of `NSFetchedResultsController` is typically places as a property inside a `UITableViewController`
- The `NSFetchedResultsController` is initialized with four parameters:
  1. a fetch request (containing at least one sort descriptor to order the results)
  2. a managed object context used to execute the fetch request
  3. a key path on result objects that returns the section name (used to split the results into sections; pass `nil` to generate a single section)
  4. the name of the cache file the controller should use (pass `nil` to prevent caching)
- After the creation of a `NSFetchedResultsController`, the fetch can be performed using the `performFetch:` method

Working with NSFetchedResultsController

```
self.fetchedResultsController = [[NSFetchedResultsController alloc]
                                          initWithFetchRequest:request
                                          managedObjectContext:managedObjectContext
                                            sectionNameKeyPath:nil
                                                     cacheName:nil];


NSError *error;
BOOL success = [self.fetchedResultsController performFetch:&error];
```

Implementing UITableViewDataSource methods

- The `NSFetchedResultsController` can be used to provide an implementation of the `UITableViewDataSource` protocol required methods

```
- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView{

    NSInteger sections = [[self.fetchedResultsController sections] count];
    return sections;

}

- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section{
    NSInteger rows = 0;
    if ([[self.fetchedResultsController sections] count] > 0) {
        id <NSFetchedResultsSectionInfo> sectionInfo =
                                [[self.fetchedResultsController sections] objectAtIndex:section];
        rows = [sectionInfo numberOfObjects];
    }
    return rows;
}
```

Implementing UITableViewDataSource methods

```
- (NSString *)tableView:(UITableView *)tableView titleForHeaderInSection:(NSInteger)section{
  return [[[self.fetchedResultsController sections] objectAtIndex:section] name];
}

- (NSInteger)tableView:(UITableView *)tableView
sectionForSectionIndexTitle:(NSString *)title
                    atIndex:(NSInteger)index{
  return [self.fetchedResultsController sectionForSectionIndexTitle:title atIndex:index];
}

- (NSArray *)sectionIndexTitlesForTableView:(UITableView *)tableView{
    return [self.fetchedResultsController sectionIndexTitles];
}
```

Implementing UITableViewDataSource methods

- The `objectAtIndexPath:` method can be used to retrieve the object inside a `NSFetchedResultsController` for a given index path
- It is how to get the object inside `tableView:cellForRowAtIndexPath:`

```
- (UITableViewCell *)tableView:(UITableView *)tableView
          cellForRowAtIndexPath:(NSIndexPath *)indexPath{

    UITableViewCell *cell = [self.tableView dequeueReusableCellWithIdentifier:@"identifier"];

    Teacher *teacher = [self.fetchedResultsController objectAtIndexPath:indexPath];

    /* configure the cell */

    return cell;

}
```

NSFetchedResultsControllerDelegate

- An **NSFetchedResultsController** has a delegate that gets notified when the controller's fetch results have been changed due to an add, remove, move, or update operations
- Be careful: a large number of modifications might occur simultaneously, so responding to all the changes separately might be computationally expensive
- Protocol methods:

```
- (void)controllerWillChangeContent:(NSFetchedResultsController *)controller

- (void)controller:(NSFetchedResultsController *)controller
  didChangeSection:(id <NSFetchedResultsSectionInfo>)sectionInfo
        atIndex:(NSUInteger)sectionIndex
   forChangeType:(NSFetchedResultsChangeType)type

- (void)controller:(NSFetchedResultsController *)controller
   didChangeObject:(id)anObject
      atIndexPath:(NSIndexPath *)indexPath
    forChangeType:(NSFetchedResultsChangeType)type
     newIndexPath:(NSIndexPath *)newIndexPath

- (void)controllerDidChangeContent:(NSFetchedResultsController *)controller
```

Mobile Application Development

Lecture 22
Core Data