

# App-Entwicklung für iOS und OS X

SS 2016  
Stephan Gimbel



# Core Data

---

## Datenbank

Manchmal müssen wir große Datenmengen speichern oder komplexe Abfragen durchführen.

Allerdings soll dies objekt-orientiert sein!

## Lösung ist Core Data

Objekt-orientierte Datenbank.

Sehr mächtiges Framework in iOS (wir kratzen hier nur an der Oberfläche)

Möglichkeit einen Objekt-Graphen zu erstellen, der in einer Datenbank liegt  
Normalerweise mittels SQL (aber auch mit XML oder nur im Speicher selbst).

## Wie funktioniert's?

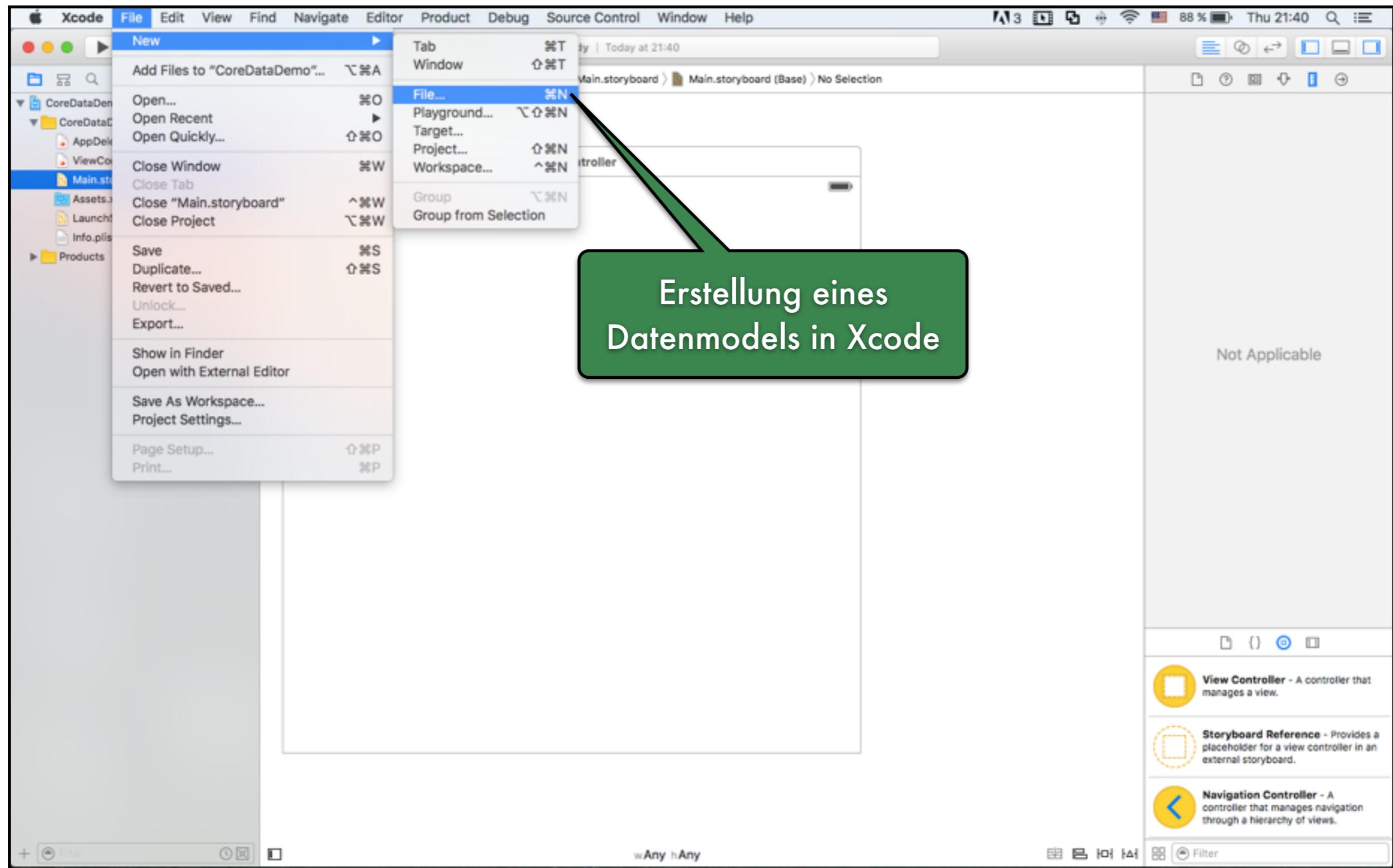
Erstellung eines visuellen Mappings (mittels Xcode) zwischen Datenbank und Objekten.

Erstellen und Abfragen von Objekten mittels objekt-orientierter API.

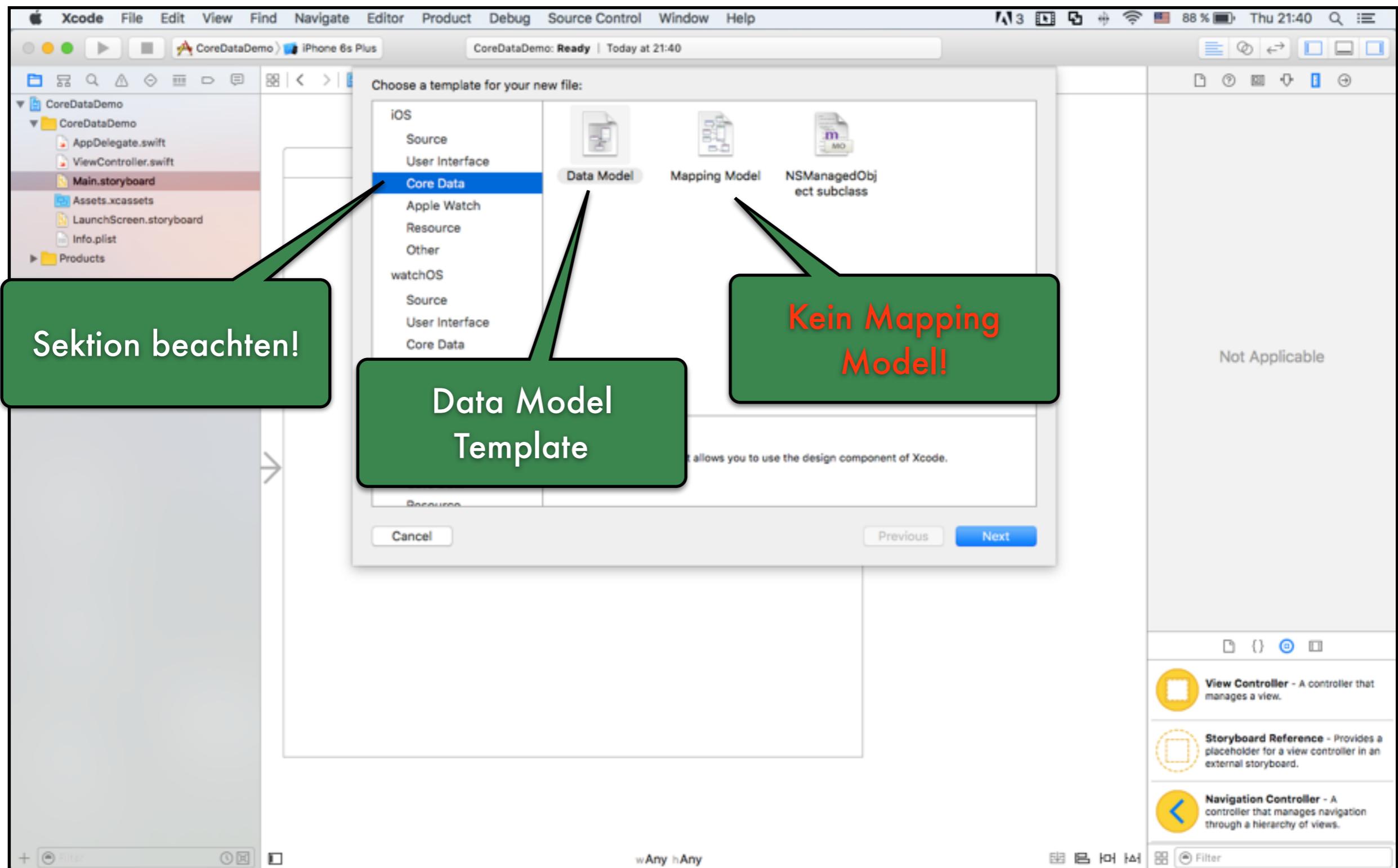
Zugriff auf Datenbank-Table mittels vars dieser Objekte.

Zuerst erstellen wir das Mapping...

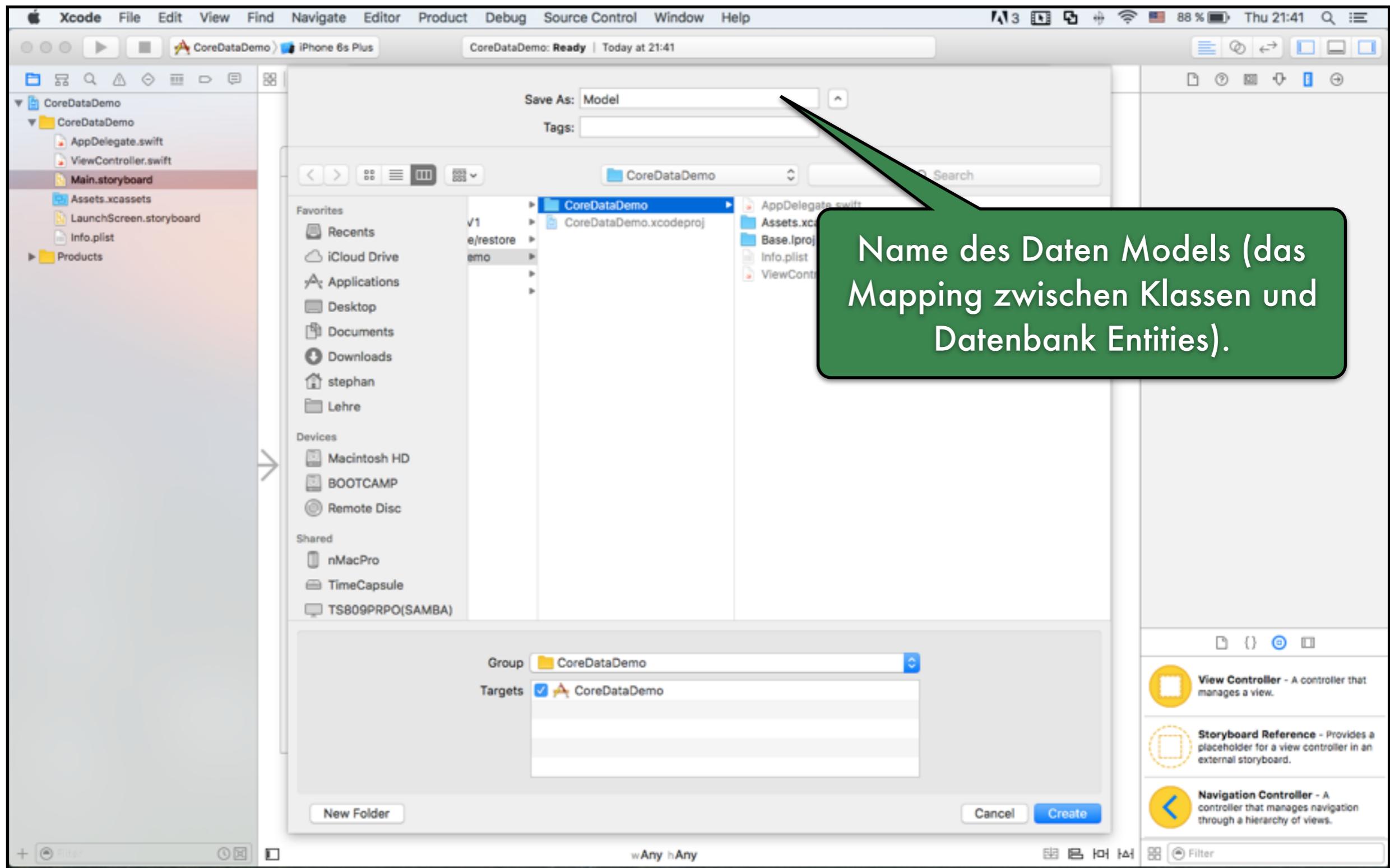
# Core Data



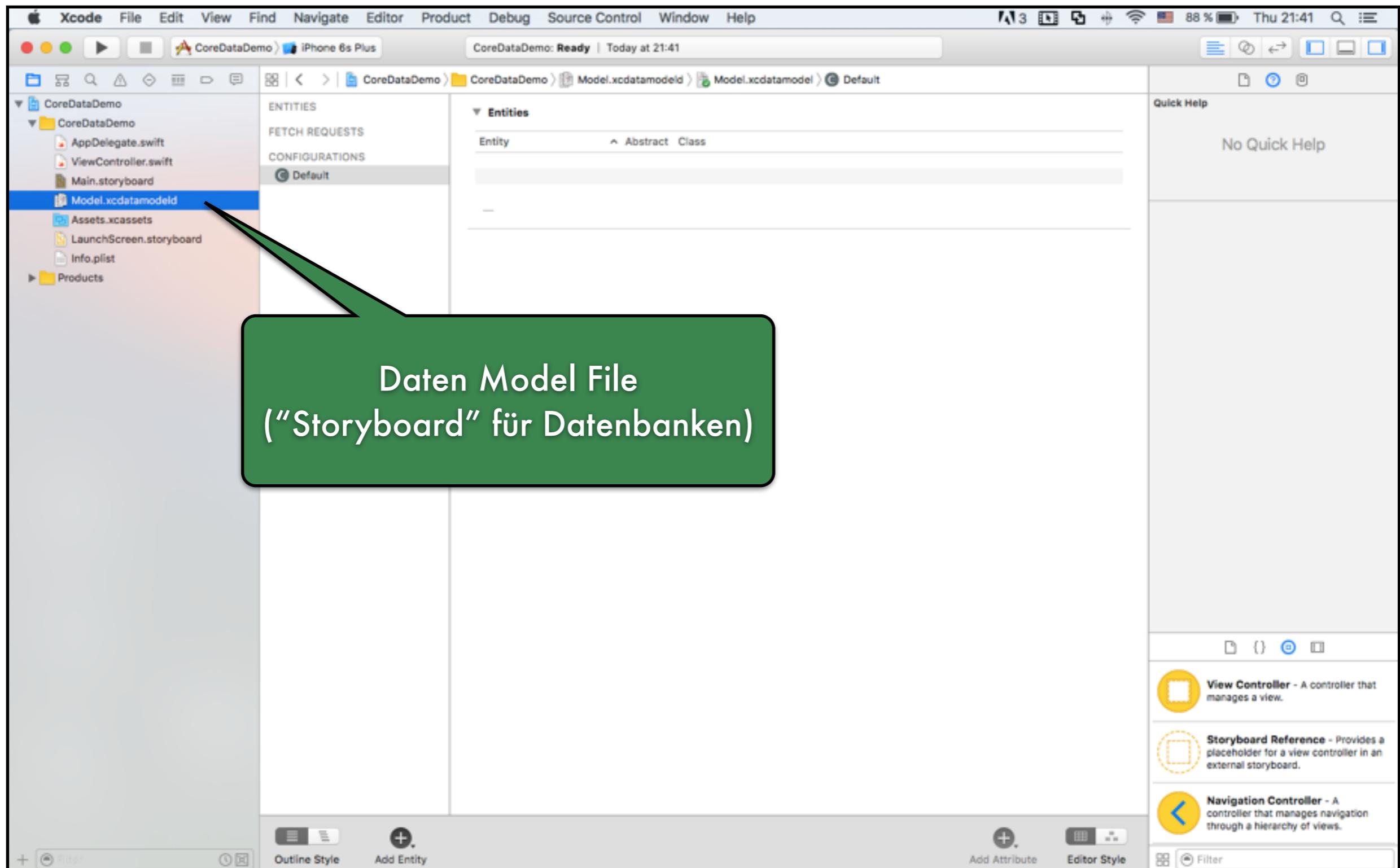
# Core Data



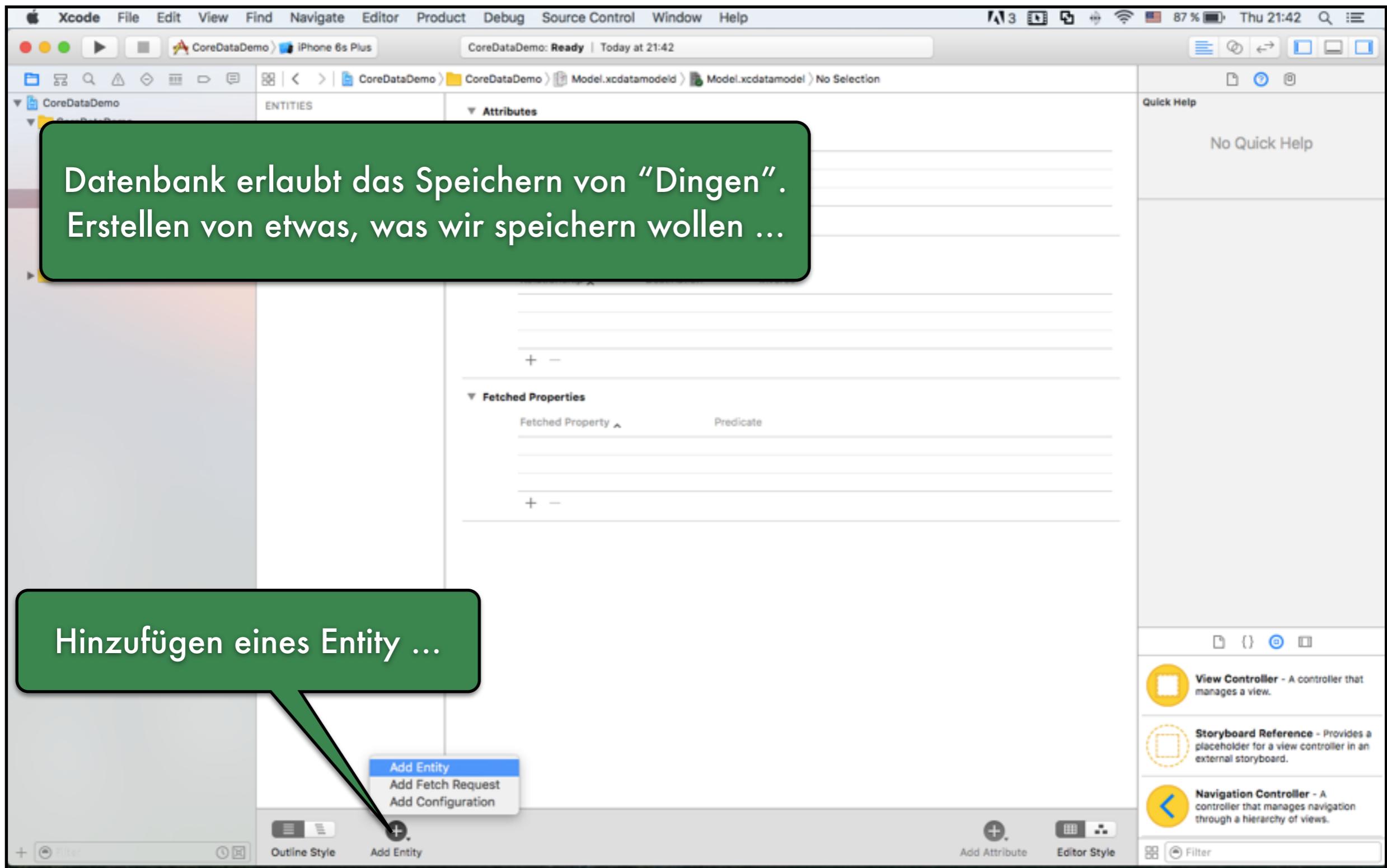
# Core Data



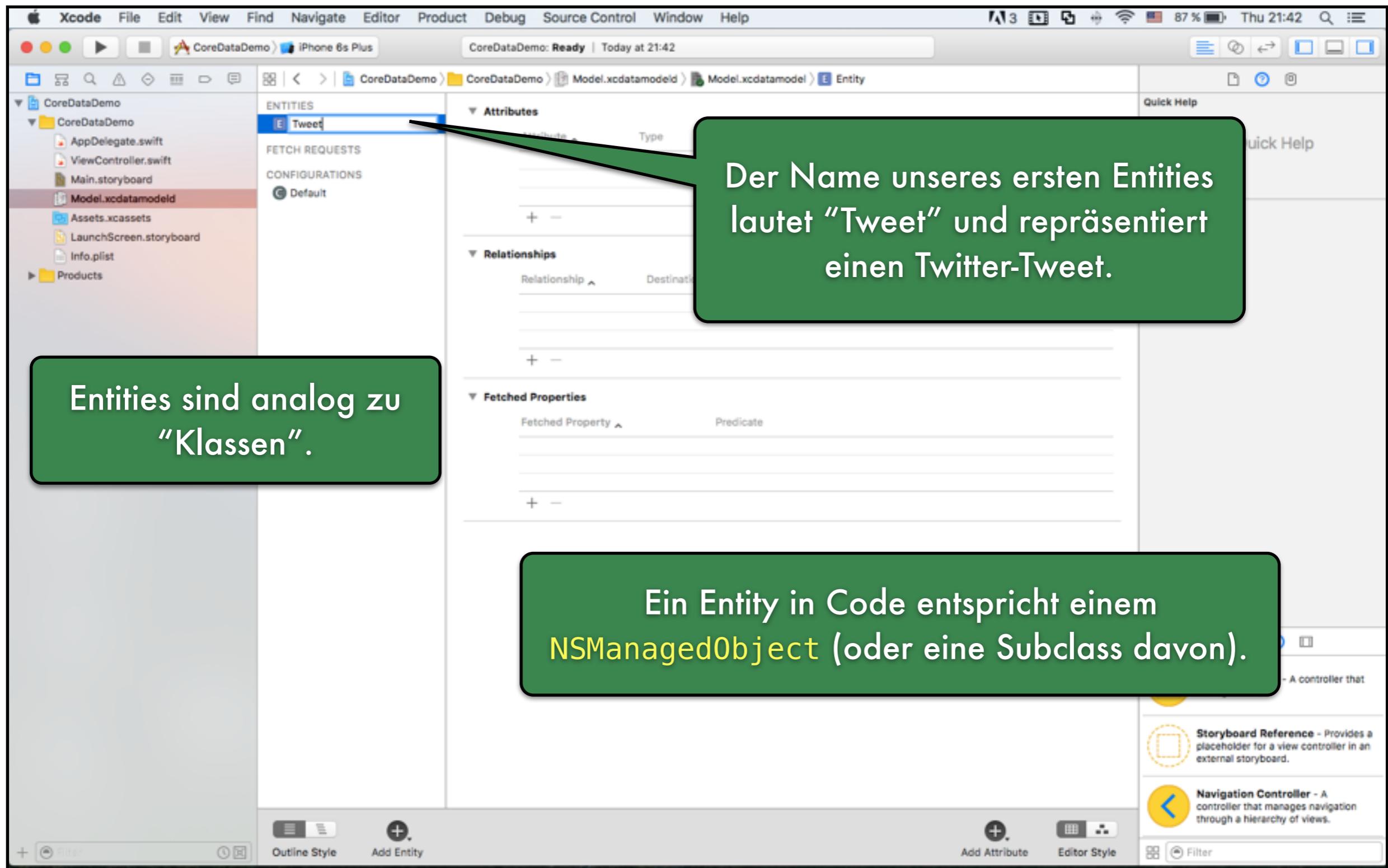
# Core Data



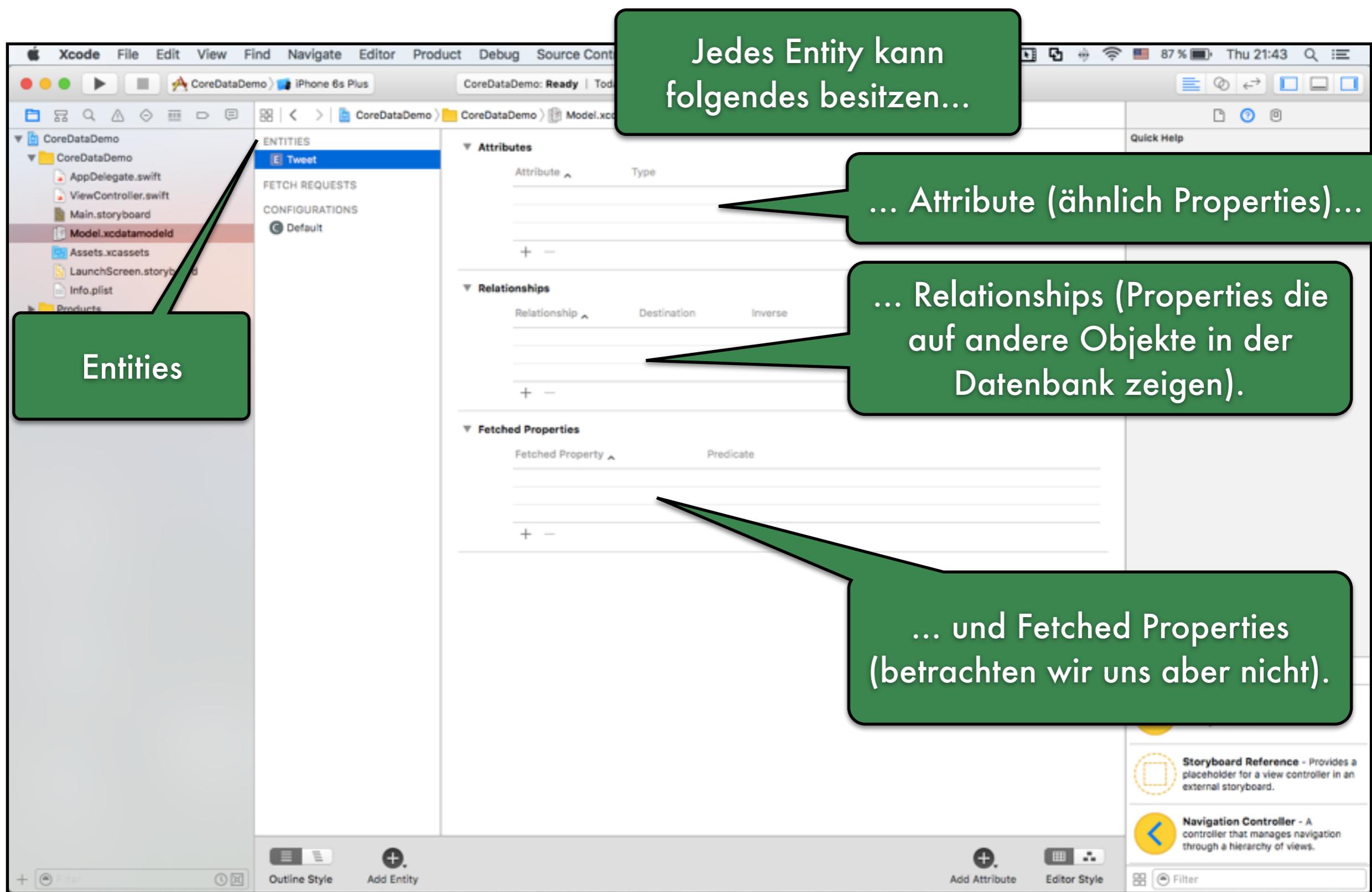
# Core Data



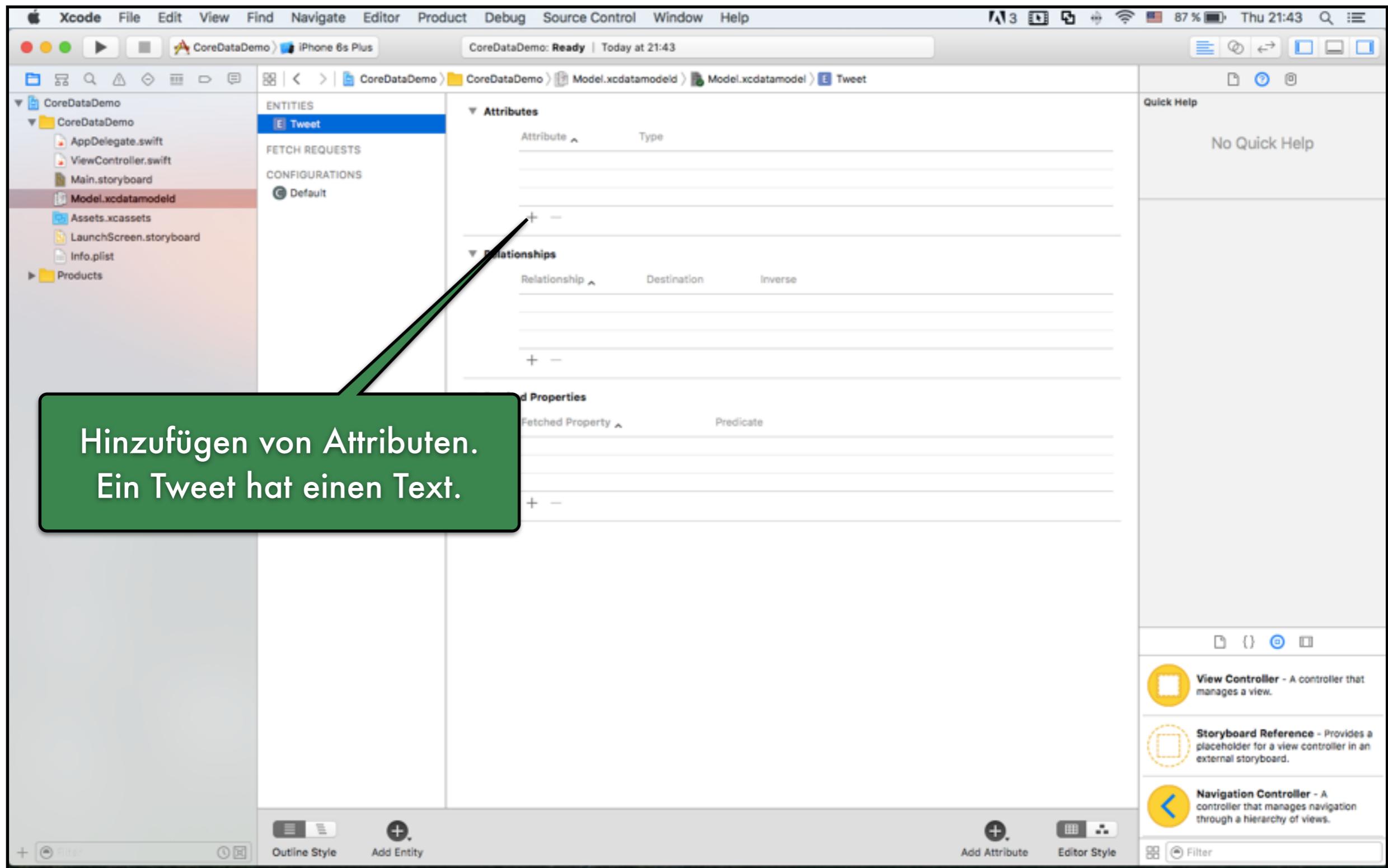
# Core Data



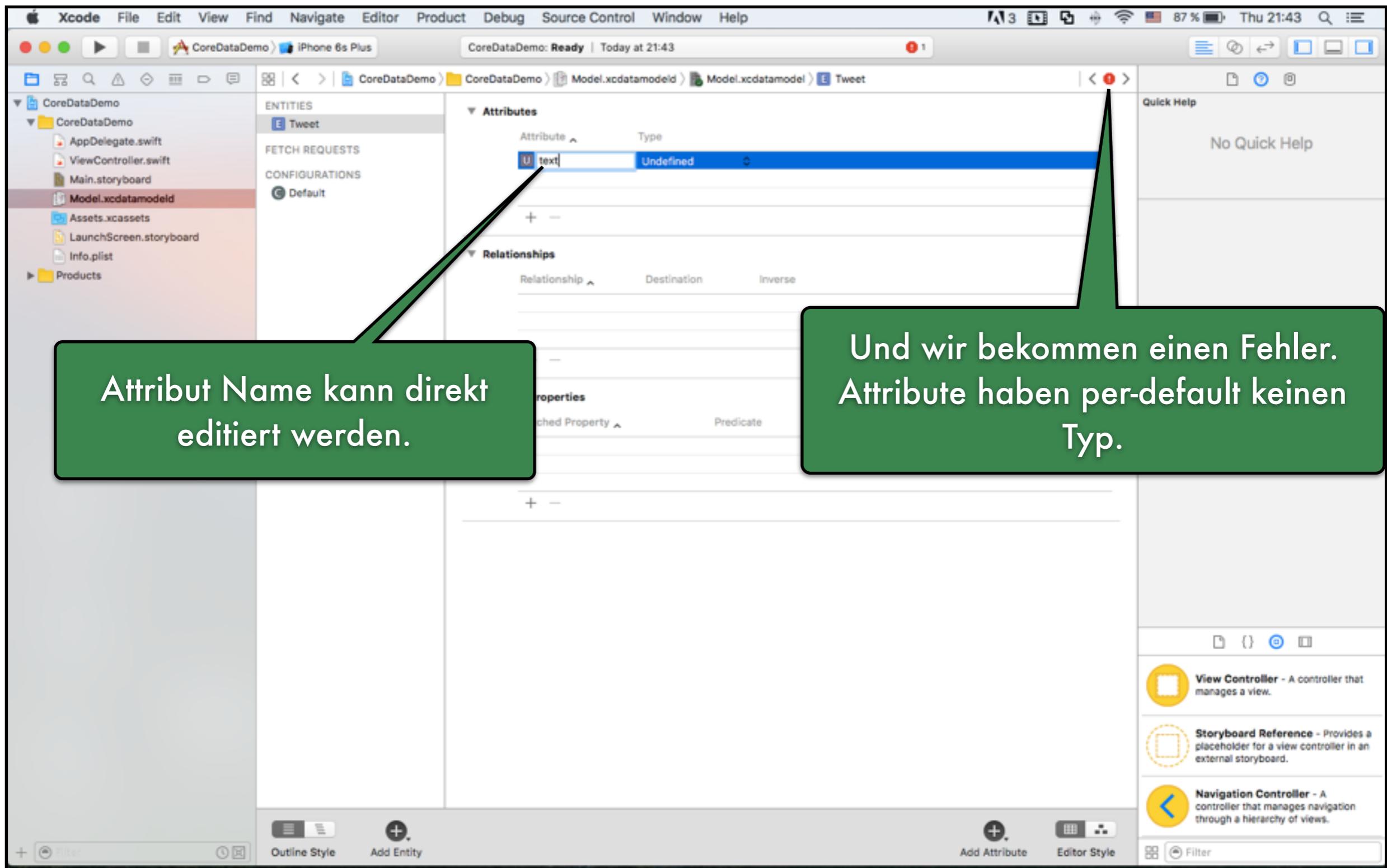
# Core Data



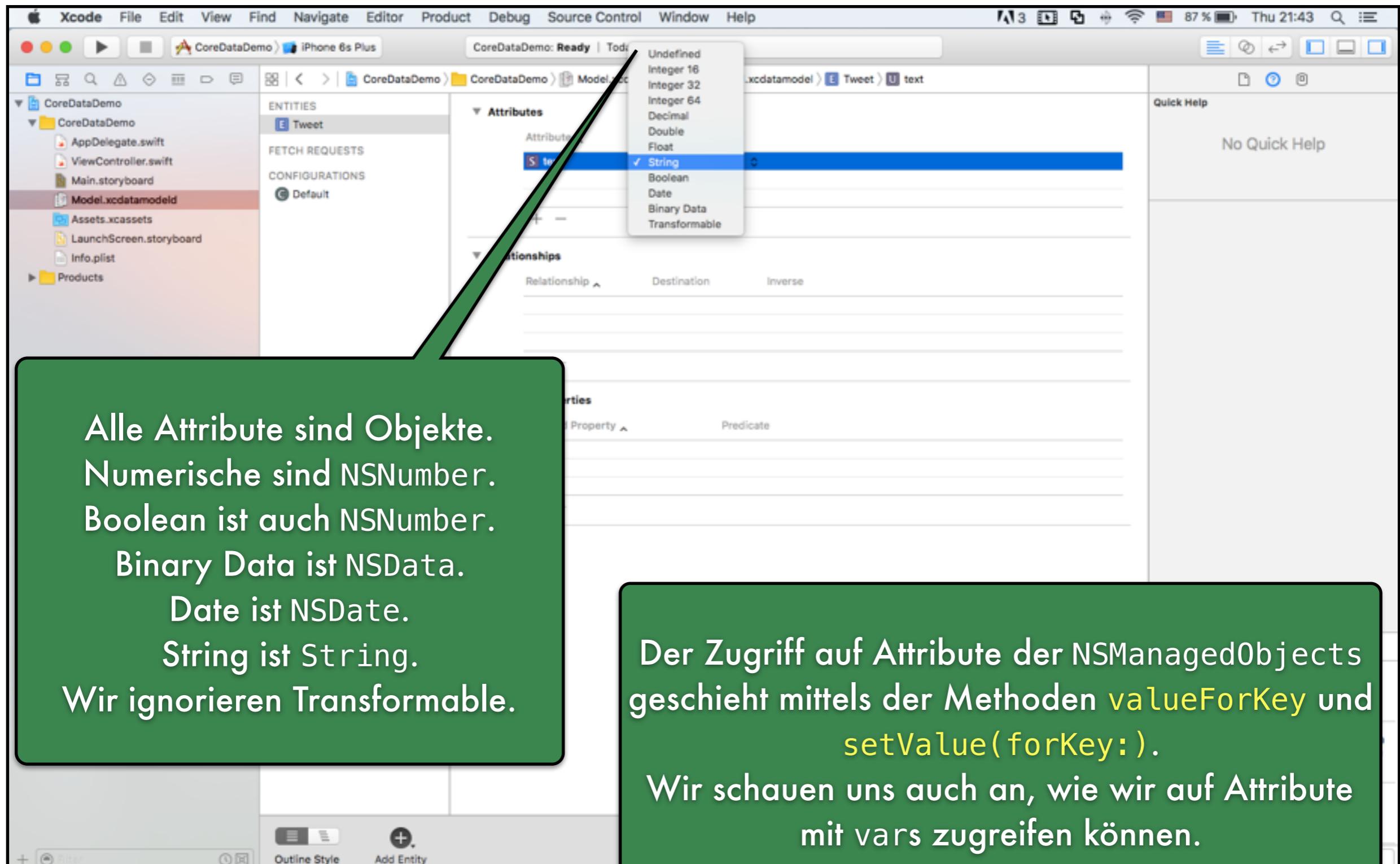
# Core Data



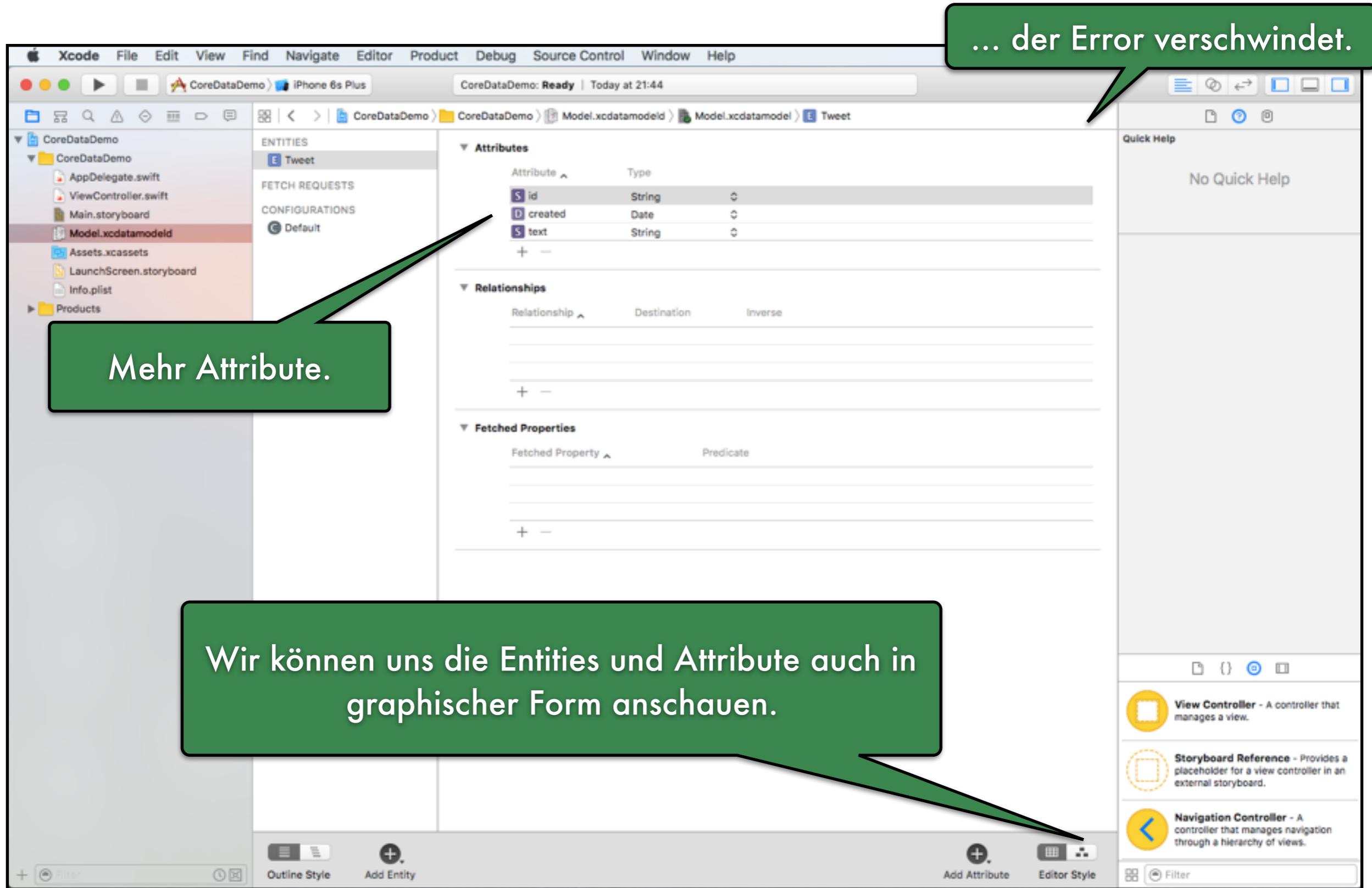
# Core Data



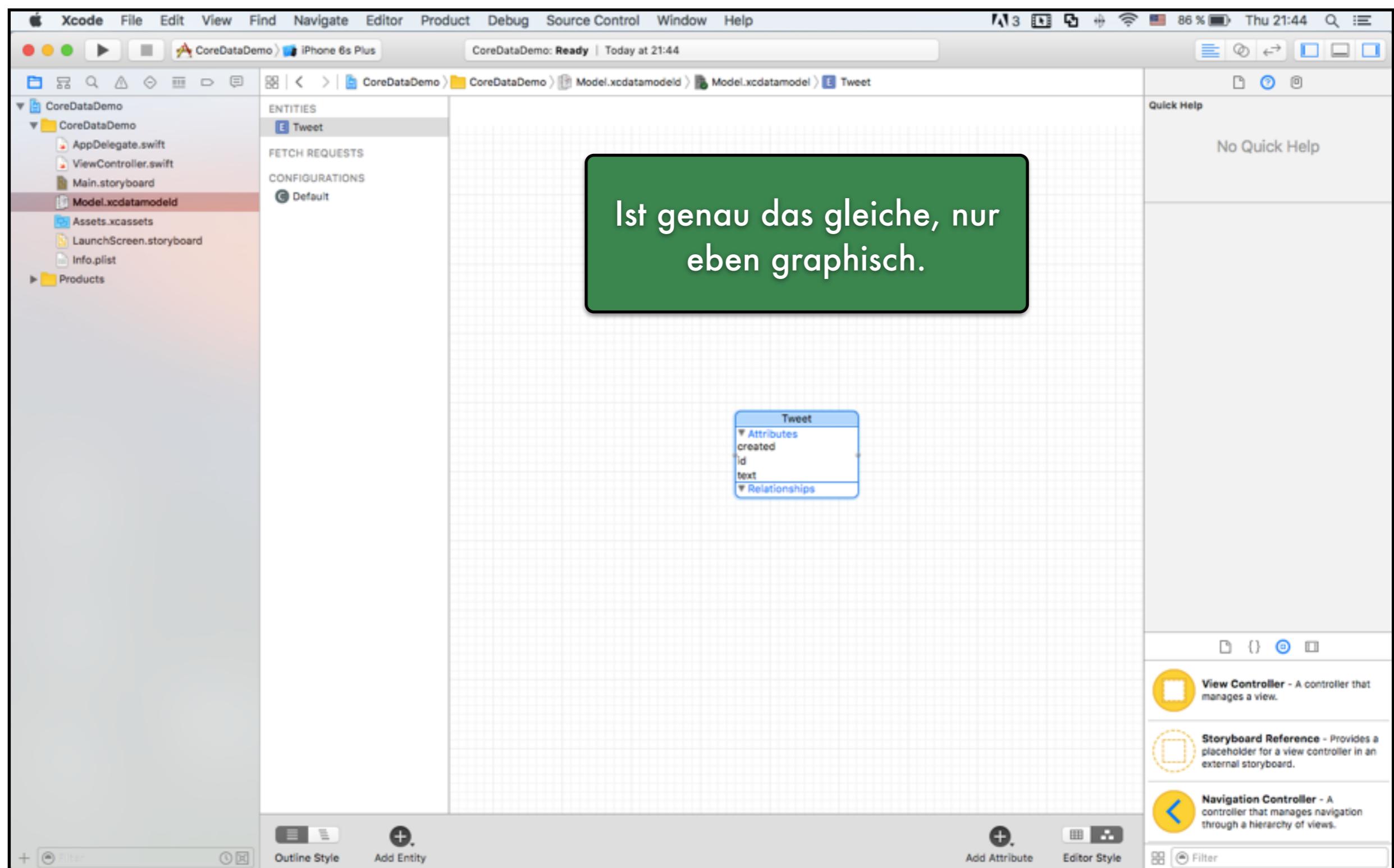
# Core Data



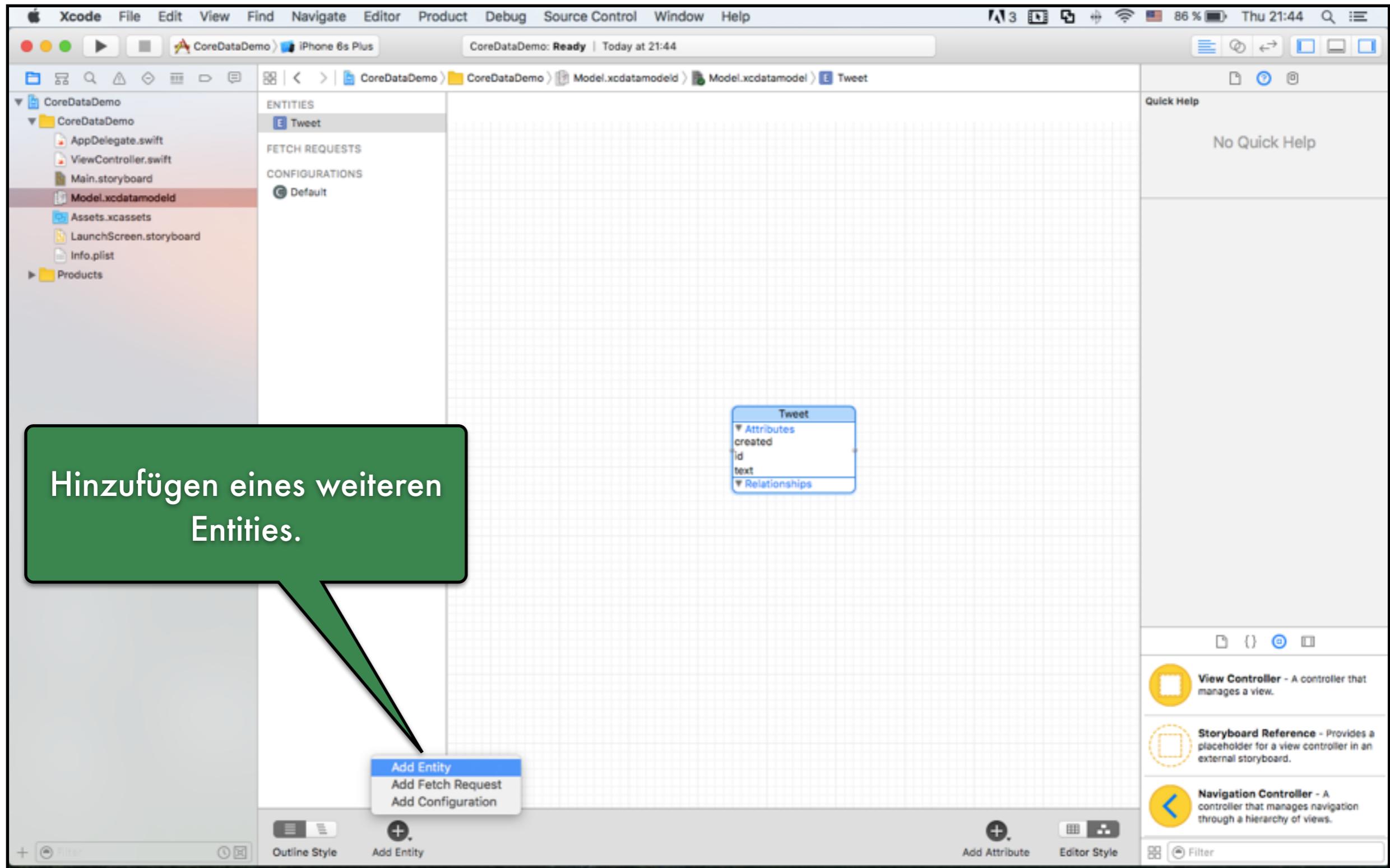
# Core Data



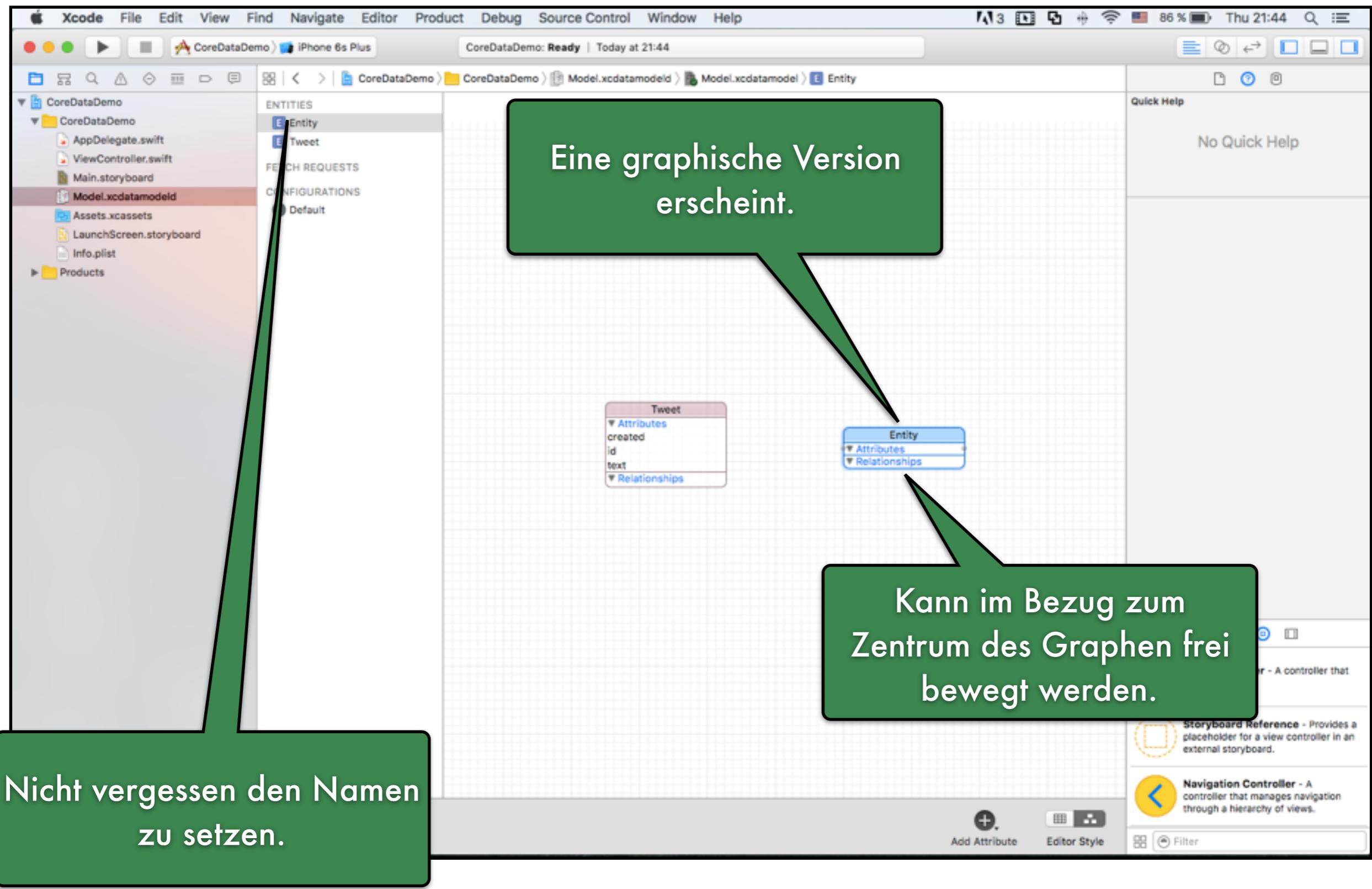
# Core Data



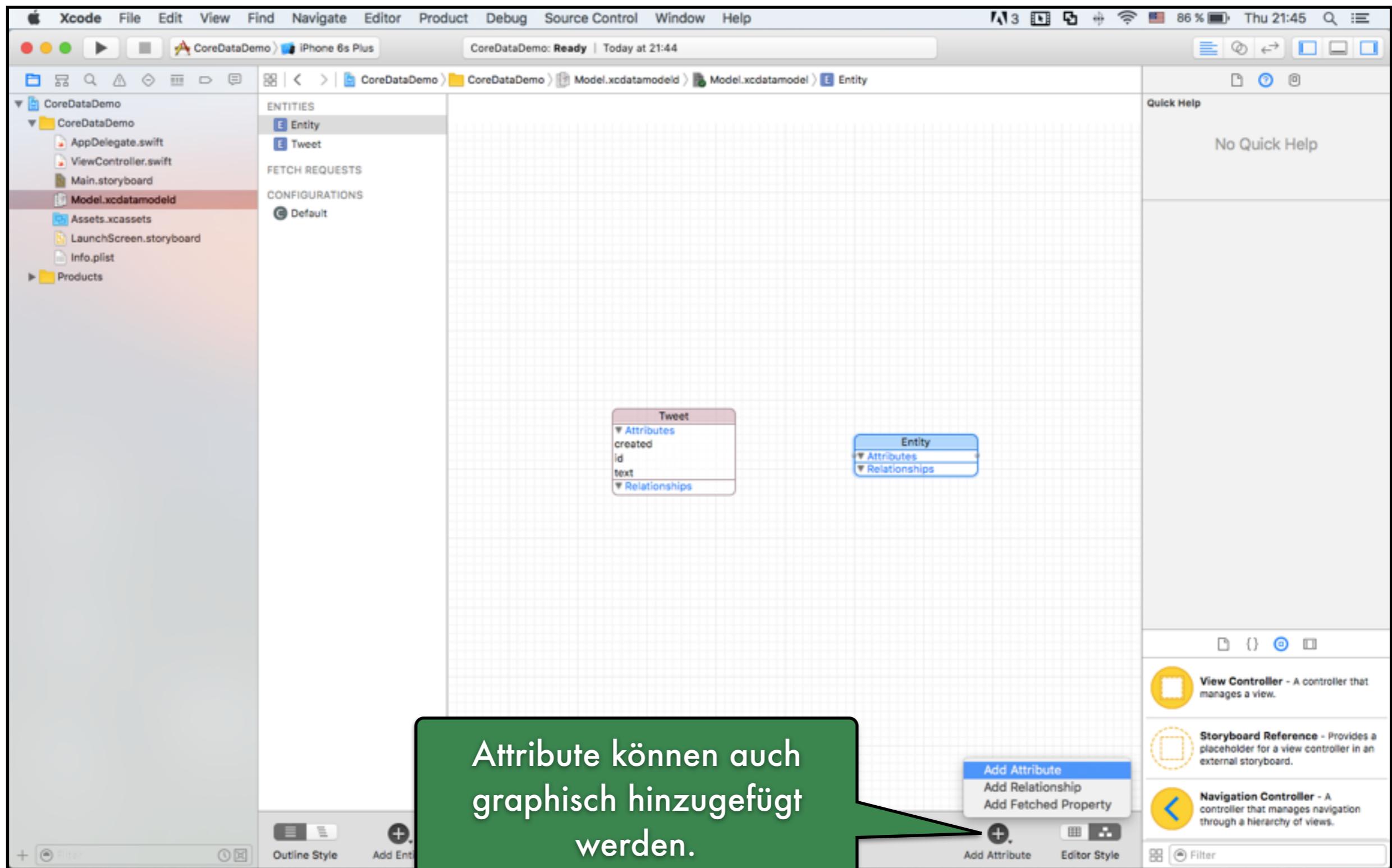
# Core Data



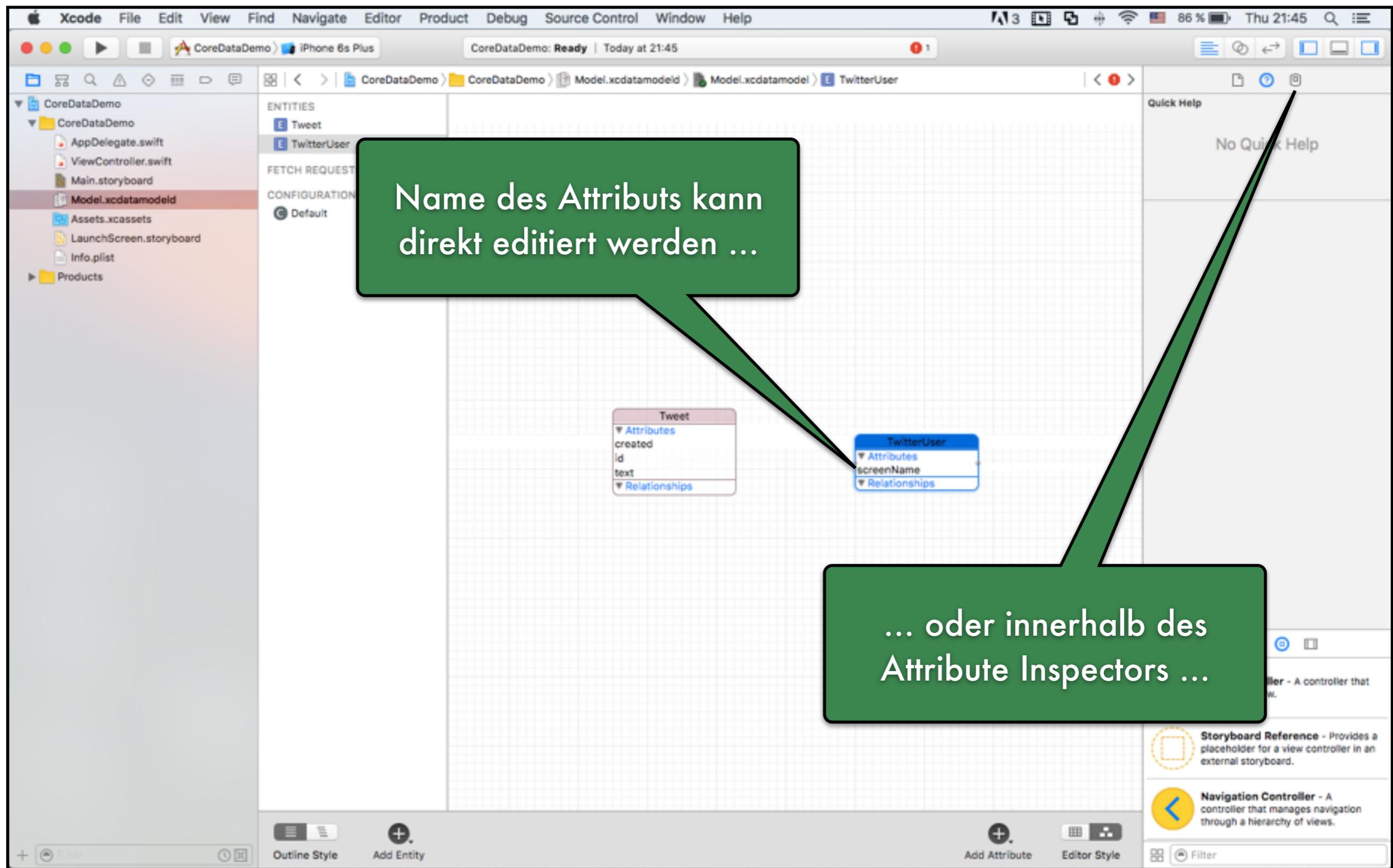
# Core Data



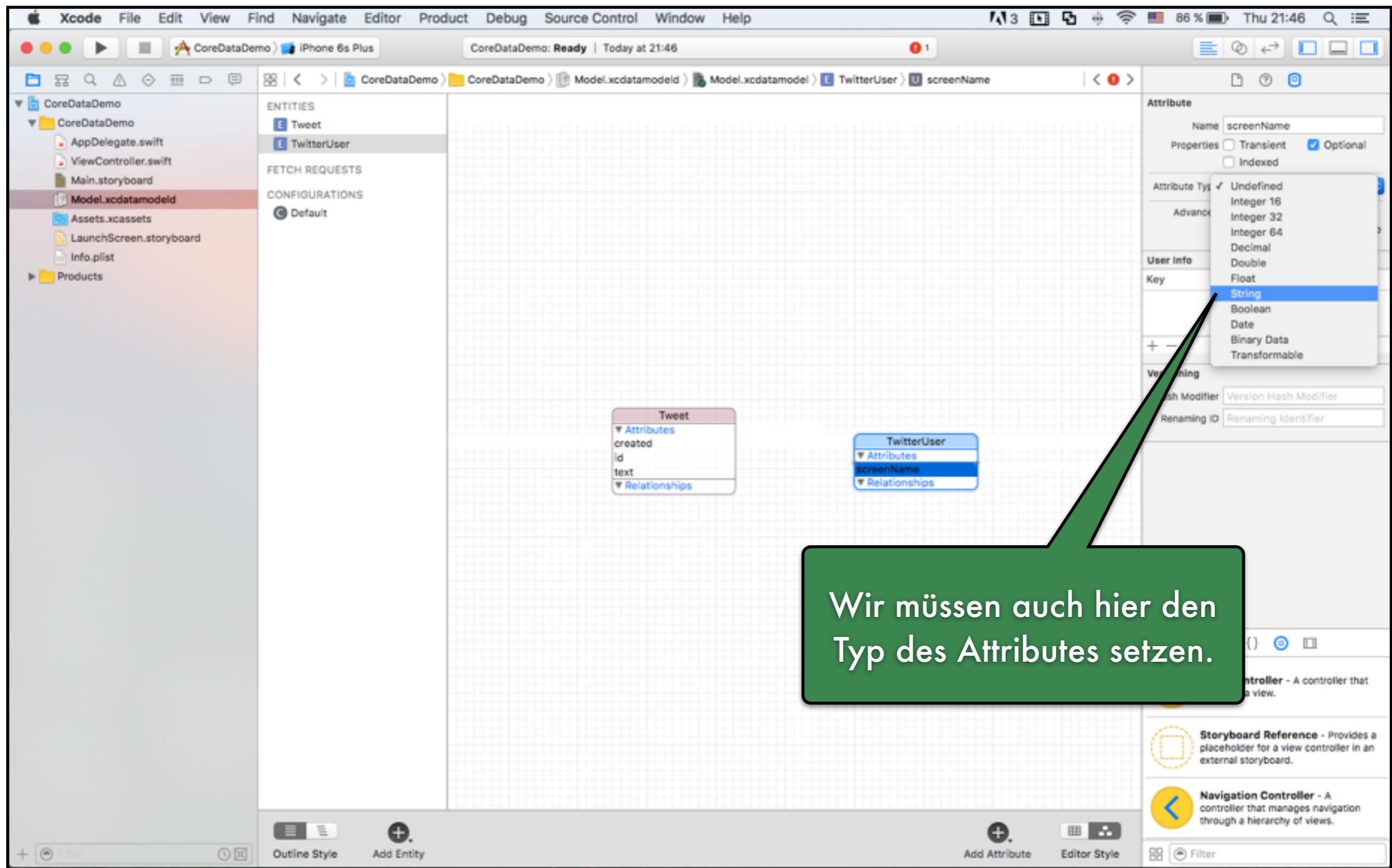
# Core Data



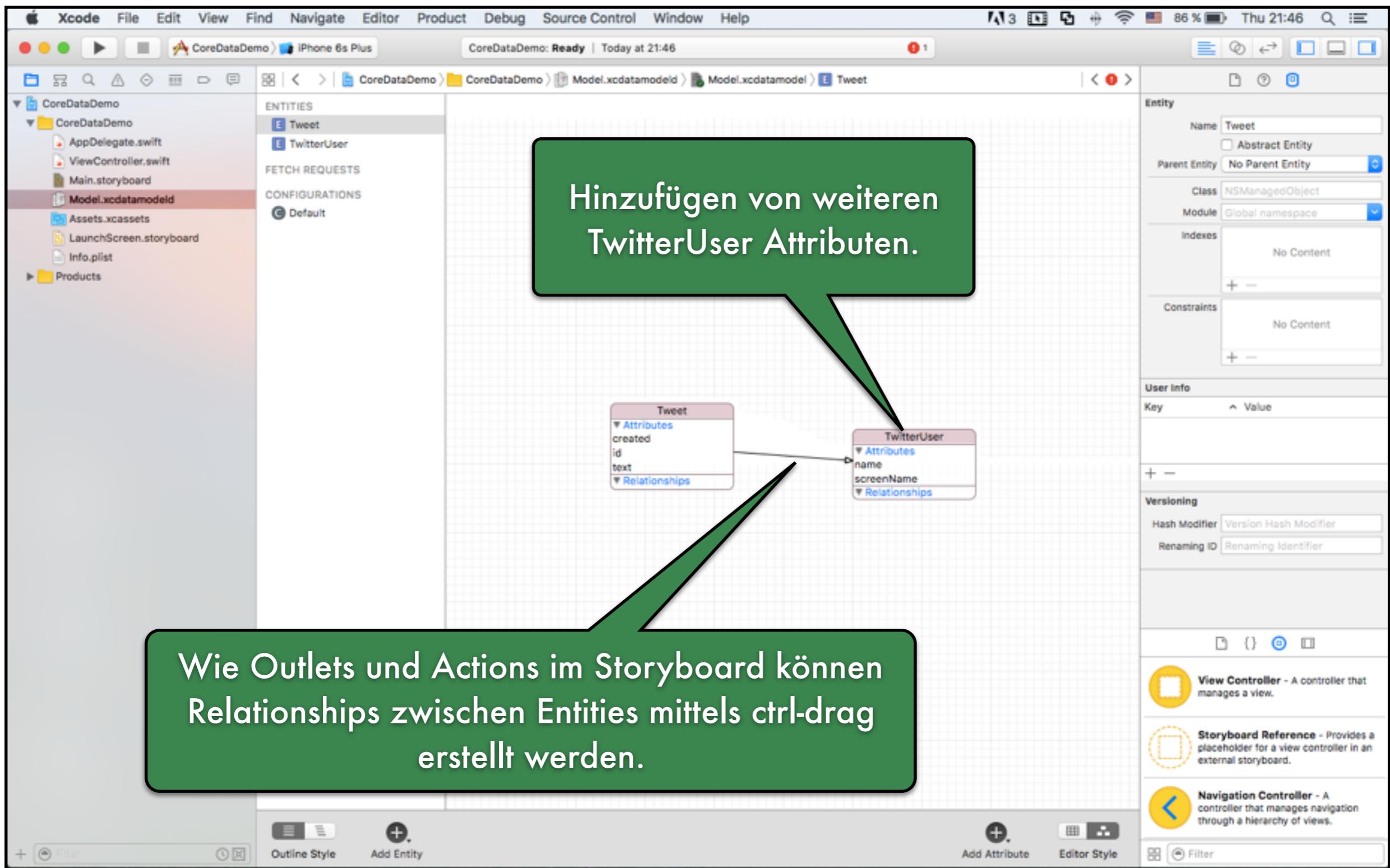
# Core Data



# Core Data



# Core Data



# Core Data

Eine Relationship ist analog zu einem Pointer zu einem anderen Objekt. (oder NSSet von anderen Objekten).

The screenshot shows the Xcode interface with the Core Data Model Editor open. The left sidebar shows project files including Model.xcdatamodeld. The main area displays two entities: Tweet and TwitterUser. The Tweet entity has attributes created, id, and text, and a relationship named newRelationship. The TwitterUser entity has attributes name and screenName, and a relationship named newRelationship. A double-headed arrow connects the newRelationship attributes of both entities, indicating a bidirectional relationship. The right side of the screen shows the Entity inspector for the selected Tweet entity, which is currently set to have a Name of "Multiple Values".

# Core Data

Xcode File Edit View Find Navigate Editor Product Debug Source Control Window Help

CoreDataDemo Ready | Today at 21:47

CoreDataDemo CoreDataDemo Model.xcdatamodeld Model.xcdatamodeld Tweet tweeter

Relationship Name tweeter Properties Transient Optional Destination TwitterUser 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

View Controller - A controller that manages a view.

Storyboard Reference - Provides a placeholder for a view controller in an external storyboard.

Navigation Controller - A controller that manages navigation through a hierarchy of views.

ENTITIES

Entities:

- E Tweet
- E TwitterUser

FETCH REQUESTS

CONFIGURATIONS

Default

Relationships:

- Tweet → tweeter (To One, Optional)
- TwitterUser ← newRelationship (Inverse, To One)

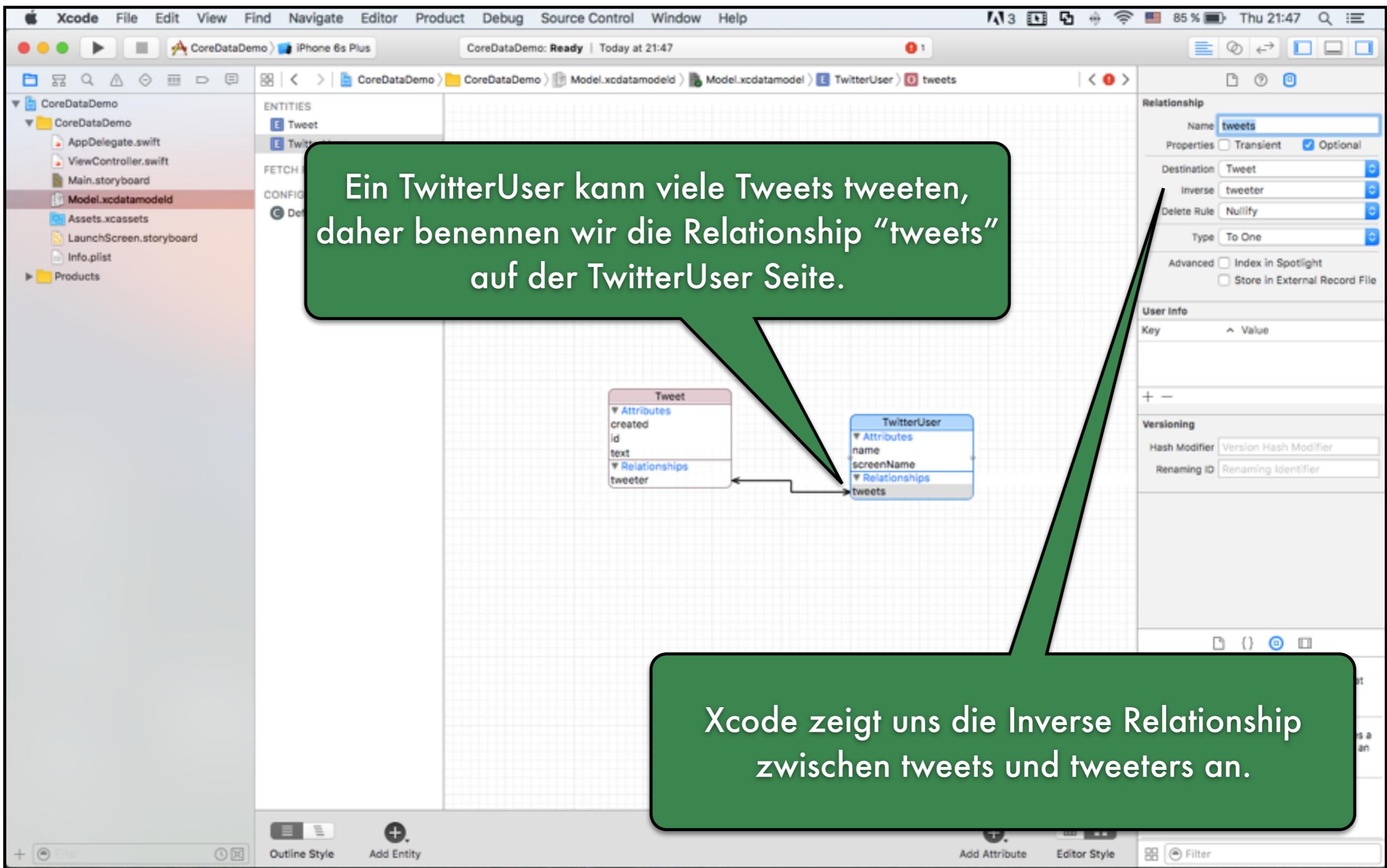
Attributes:

- Tweet: created, id, text
- TwitterUser: name, screenName

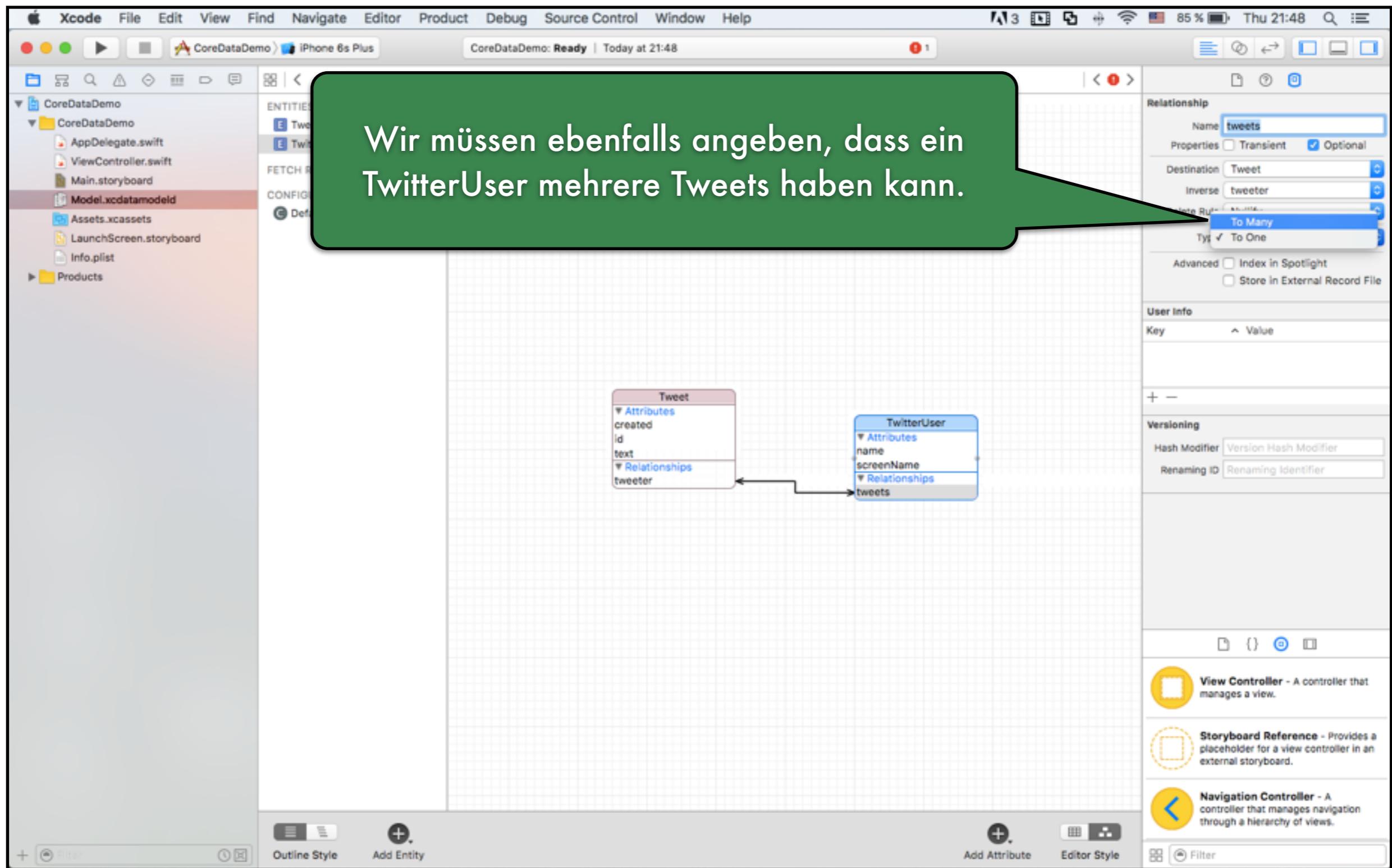
Annotations:

- A callout bubble points from the "tweeter" relationship in the Tweet entity to the text "... benennen der Relationship als 'tweeter' auf der Tweet Seite."
- A callout bubble points from the "tweeter" relationship in the Relationship inspector to the text "Die Relationship für einen Tweet zu einem TwitterUser ist der ‘tweeter’ des Tweets ..."

# Core Data



# Core Data



Wir müssen ebenfalls angeben, dass ein TwitterUser mehrere Tweets haben kann.

# Core Data

The screenshot shows the Xcode interface with the Core Data Model Editor open. The project navigation sidebar on the left lists files like AppDelegate.swift, ViewController.swift, Main.storyboard, Model.xcdatamodeld, Assets.xcassets, LaunchScreen.storyboard, and Info.plist. The main editor area displays two entities: Tweet and TwitterUser. The Tweet entity has attributes created, id, and text, and a relationship named tweeter pointing to TwitterUser. The TwitterUser entity has attributes name and screenName, and a relationship named tweets pointing back to Tweet. A green callout points from the 'tweeter' relationship to a text box explaining it represents a "to many" relationship. Another green callout points from the 'tweets' relationship to a text box explaining its delete rule. A third green callout points from the 'tweets' relationship to a text box explaining its type.

Der Typ der Relationship in Code ist ein NSManagedObject (oder eine Subclass davon).

Der Doppelpfeil bedeutet eine "to many" Relationship (aber nur in diese Richtung)

Die Delete-Rule gibt an was mit den auf die gezeigten Tweets passiert, wenn der TwitterUser gelöscht wird.

Nullify bedeutet "setze den tweeter Pointer auf nil".

# Core Data

---

Viele andere Funktionen für Core Data

Wir beschränken uns aber auf das Erstellen von Entities, Attribute und Relationships.

Wir können wir darauf in Code zugreifen?

Wir benötigen einen `NSManagedObjectContext`.

Der Knotenpunkt um den sich alle Core Data Aktivität dreht.

Wir können wir diesen erhalten?

Zwei Möglichkeiten...

1. Klick auf “Use Core Data” beim Erstellen des Projektes
2. Erstellen eines `UIManagedDocument` und nach dessen `managedObjectContext` fragen (eine var).

# Core Data

---

Sharing eines globalen NSManagedObjectContext im AppDelegate

Das Klicken auf “Use Core Data” beim Erstellen des Projektes fügt Code zum AppDelegate hinzu.

Das wichtigste ist dabei die `managedObjectContext` var.

Wir können auf die AppDelegate's `managedObjectContext` var so zugreifen...

```
(UIApplication.sharedApplication().delegate as! AppDelegate).managedObjectContext
```

Tipp: Wenn wir ein bestehendes Projekt haben, einfach ein neues Projekt erstellen und den AppDelegate Code kopieren.

Nicht nur die `managedObjectContext` var, sondern alle Methoden mit einer Abhängigkeit dazu.

# Core Data

---

## UIManagedDocument

Erbt von UIDocument welches viele Mechanismen zum Speichermanagement zur Verfügung stellt.

Wenn wir UIManagedDocument nutzen, können wir sehr einfach auf iCloud Support umsteigen.

UIManagedDocument ist eine Art Container für die Core Data Datenbank.

## Erstellen eines UIManagedDocument

Zuerst müssen wir eine URL zu einem File erstellen in dem das Document gespeichert wird.

Dies verlangt Verständnis des File-Systems, welches wir bisher noch nicht behandelt haben.

In Code wie folgt...

```
let fm = NSFileManager.defaultManager()
if let docsDir = fm.URLsForDirectory(.DocumentDirectory, inDomains: .UserDomain).first {
    let url = docsDir.URLByAppendingPathComponent("MyDocumentName")
    let document = UIManagedDocument(fileURL: url)
}
```

Dies erstellt die UIManagedDocument Instanz, öffnet oder erstellt aber nicht das darunter liegende File.

# Core Data

---

## Öffnen oder Erstellen eines UIManagedDocument

Bevor wir ein UIManagedDocument nutzen können, müssen wir zuerst Prüfen ob es geöffnet ist.

Wenn es bereits geöffnet ist (.Normal Status), können wir den managedObjectContext direkt nutzen.

```
if document.documentState == .Normal { /* use it */ }
```

Wenn es geschlossen ist (.Closed) ...

```
if document.documentState == .Closed { /* open or create */ }
```

... dann müssen wir es öffnen oder erstellen.

Um dies zu tun, zuerst prüfen ob das unterliegende File des UIManagedDocument's auf der "disk" existiert...

```
if let path = fileURL.path, let fileExists =  
    FileManager.defaultManager().fileExistsAtPath(path) { ... }
```

Wenn es existiert, das Document öffnen...

```
document.openWithCompletionHandler { (success: Bool) in /* use MOC */ }
```

Wenn es nicht existiert, das Document erstellen...

```
document.saveToURL(document.fileURL, forSaveOperation: .ForCreating)  
{ success in ... }
```

# Core Data

---

All dies ist asynchron!

Öffnen oder erstellen des Documents dauert vielleicht eine Weile.

Und wir wollen (wie immer) den Main Thread nicht blockieren.

Jedoch wird der Block eventuell zurück im Main Thread ausgeführt.

Andere documentStates

- SavingError (success ist NO im Completion Handler).
- EditingDisabled (temporär, noch mal versuchen).
- InConflict (z.B. da ein anderes Device eine Änderung via iCloud gemacht hat).

Schauen wir uns vielleicht später noch an (wenn wir Zeit haben).

Speichern des Documents

UIManageDocument hat ein AUTOSAVE für sich selbst.

Wenn wir jedoch, aus welchen Gründen auch immer, manuell speichern wollen (asynchron) ...

```
document.saveToURL(document.fileURL, forSaveOperation:.ForOverwriting) { success in ... }
```

Fast identisch zum Erstellen (nur .ForOverwriting ist anders).

Dies ist eine UIKit Klasse, also muss diese Methode in der Main Queue aufgerufen werden.

# Core Data

---

## Schließen des Documents

Wird automatisch geschlossen, wenn es keine **strong** Pointer mehr dazu gibt.

Kann aber explizit mit einer asynchronen Methode geschlossen werden...

```
document.closeWithCompletionHandler { success in ... }
```

## Was machen wir nun mit dem NSManagedObjectContext?

Wir haben ihn erhalten aus einer offenen `UIManagedDocument's managedObjectContext` var.

Oder wir haben ihn aus dem AppDelegate erhalten mit dem Code aus dem Erstellen eines neuen Core Data Projektes.

Nun wollen wir ihn nutzen zum einfügen/löschen von Objekten in der Datenbank oder zur Abfrage von Objekten.

# Core Data

---

## Einfügen von Objekten in die Datenbank

```
let moc = aDocument.managedObjectContext // oder vom AppDelegate  
let tweet: NSManagedObject =  
    NSEntityDescription.insertNewObjectForEntityForName("Tweet",  
        inManagedObjectContext: moc)
```

Die `NSEntityDescription` Klasse gibt ein `NSManagedObject` Instanz zurück.

Alle Objekte in der Datenbank werden durch `NSManagedObjects` oder einer Subclass davon repräsentiert.

Eine Instanz von `NSManagedObject` ist eine Ausprägung eines Entities im Core Data Model (das Datenmodel welches wir in Xcode visuell erstellt haben).

Attribute von neu eingefügten Objekten sind initial nil (außer wir haben einen Default in Xcode gesetzt).

# Core Data

---

Wie greifen wir auf Attribute in einer NSManagedObject Instanz zu?

Wir können direkt über die NSKeyValueCoding Protokoll-Methoden zugreifen...

```
func valueForKey(String) -> AnyObject?  
func setValue[AnyObject?, forKey: String)
```

Wir können ebenfalls valueForKeyPath/setValue(forKeyPath:) nutzen und damit den Relationships folgen.

Der **key** ist ein Attribut-Name im Data-Mapping

Zum Beispiel "created" oder "text".

Das **value** ist was auch immer gespeichert ist (oder wird) in der Datenbank

Es ist nil, wenn noch nichts gespeichert wurde (außer wir haben in Xcode ein Default gesetzt).

Alle Values sind Objekte (Zahlen und Booleans sind NSNumber Objekte).

Binärdaten Values sind NSData Objekte.

Datum Values sind NSDate Objekte.

"To-many" gemappte Relationships sind NSSet Objekte.

Nicht-"to-many" Relationships sind andere NSManagedObjects.

# Core Data

---

Änderungen geschehen nur um Speicher, bis wir speichern

Zur Erinnerung, `UIManagedDocument` speichert automatisch.

Wenn ein Document gespeichert wird, wird der Context gespeichert und die Änderungen werden in die Datenbank geschrieben.

Vorsicht bei der Entwicklung, wenn "Stop" in Xcode gedrückt wird und Autosave noch Pending ist.

**Wir müssen explizit speichern, wenn wir kein `UIManagedDocument` nutzen**

```
let context = (UIApplication.sharedApplication as! AppDelegate).managedObjectContext  
/* Context bearbeiten */  
context.save()
```

... ist aber nicht ganz so einfach, da `save()` in `UIManagedObjectContext` einen Error werfen kann.

Wie behandeln wir Errors?

# Core Data

---

## Auch in Swift mit try-catch

Methoden die Fehler werden können, haben das Schlüsselwort **throws** am Ende.

```
func save() throws
```

Aufrufe solcher Methoden muss in einen do { } Block gesteckt werden und das Schlüsselwort **try** muss beim Aufruf verwendet werden.

```
do {
    try context.save()
} catch let error {
    // error ist etwas, was das ErrorType Protokoll implementiert, z.B. NSError
    // Normalerweise sind es enums, die einen Wert für Error Details haben
    throw error // normal den Fehler werden (sofern die Methode in der wir gerade
                // sind einen Fehler werfen kann)
}
```

Wenn wir ganz sicher sind, dass ein Aufruf keinen Fehler werfen wird, können wir dies erzwingen mit **try!** ...

```
try! context.save() // crash wenn doch ein Fehler auftritt
```

# Core Data

---

Aufruf von `valueForKey/setValue(forKey:)` ist nicht sehr elegant

Es gibt kein Type-Checking.

Und wir haben viele String Literale im Code (z.B. "created")

Was wir eigentlich wollen ist set/get mittels vars

Kein Problem... wir erstellen einfach eine Subclass von `NSManagedObject`

Diese Subclass hat vars für jedes Attribut in der Datenbank.

Wir benennen unsere Subclass mit dem gleichen Namen wie das zugehörige Entity (nicht unbedingt nötig, aber klar empfohlen).

Und wie wir uns vielleicht schon gedacht haben, generiert Xcode eine solche Klasse für uns.

# Core Data

The screenshot shows the Xcode interface with the Core Data Model Editor open. The left sidebar shows the project structure with `Model.xcdatamodeld` selected. The main editor area displays two entities: `Tweet` and `TwitterUser`. The `Tweet` entity has attributes `created`, `id`, and `text`, and a relationship `tweeter` pointing to the `TwitterUser` entity. The `TwitterUser` entity has attributes `name` and `screenName`, and a relationship `tweets` pointing back to the `Tweet` entity. A green callout box points to the entities in the list, containing the text: "Beide Entities markieren. Xcode erstellt für uns daraus NSManagedObject Subclasses." The right side of the screen shows the Entity inspector for the selected `Tweet` entity, which is currently unconfigured.

Beide Entities markieren. Xcode erstellt für uns daraus NSManagedObject Subclasses.

Entity

- Name
- Abstract Entity
- Parent Entity
- Class
- Module

Indexes

- No Content
- +  -

Constraints

- No Content
- +  -

User Info

Key	Value

+ -

Versioning

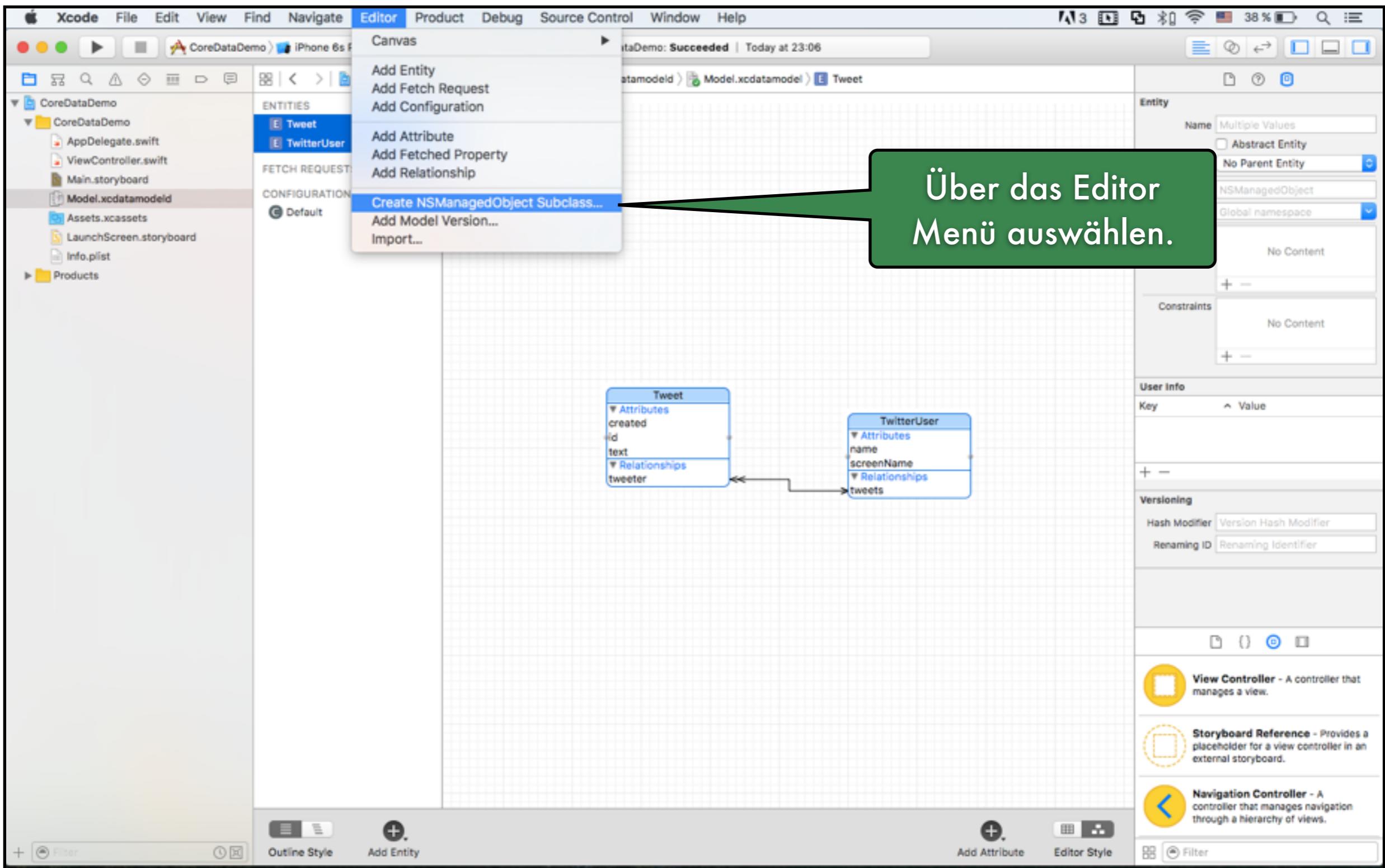
- Hash Modifier
- Renaming ID

View Controller - A controller that manages a view.

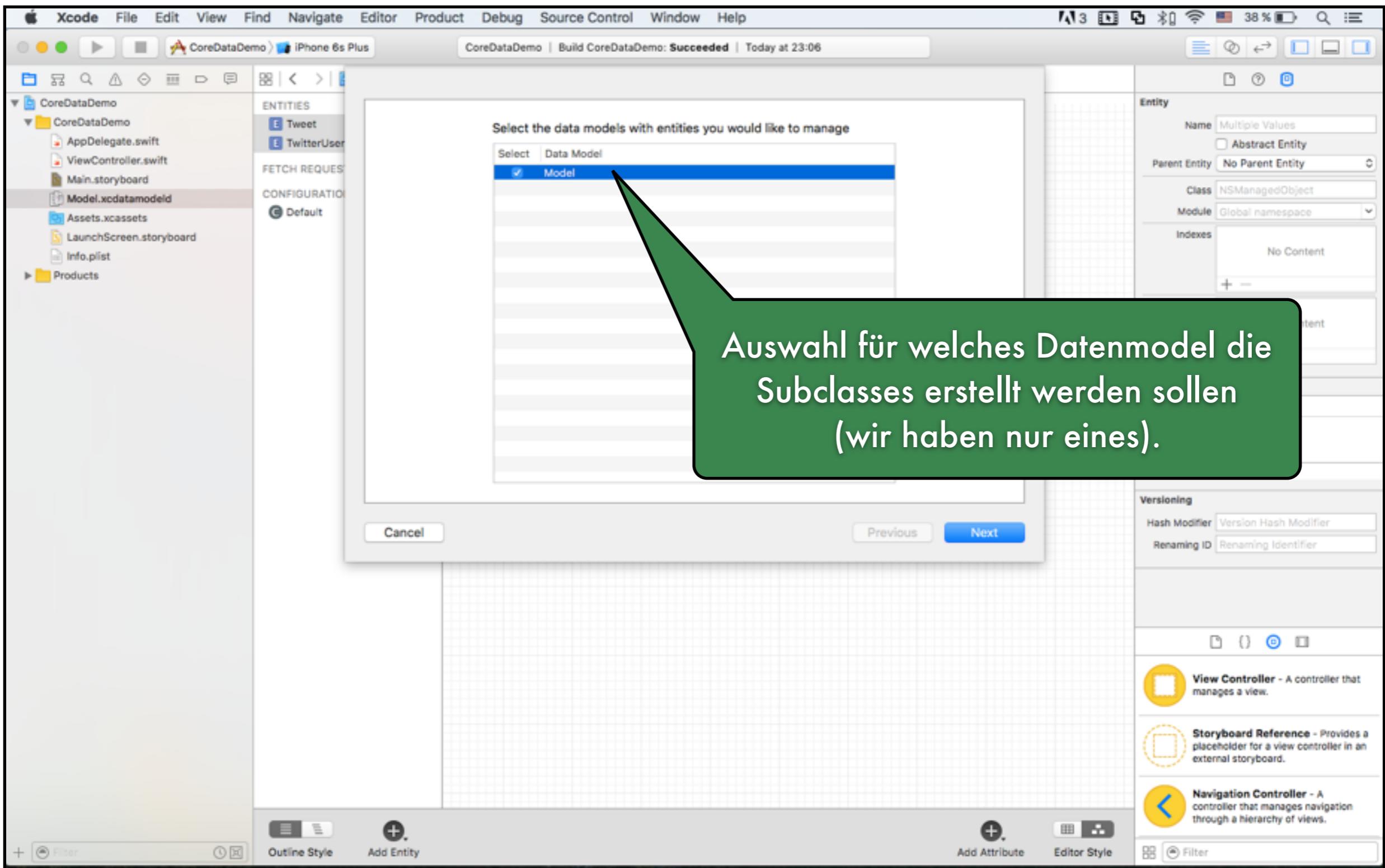
Storyboard Reference - Provides a placeholder for a view controller in an external storyboard.

Navigation Controller - A controller that manages navigation through a hierarchy of views.

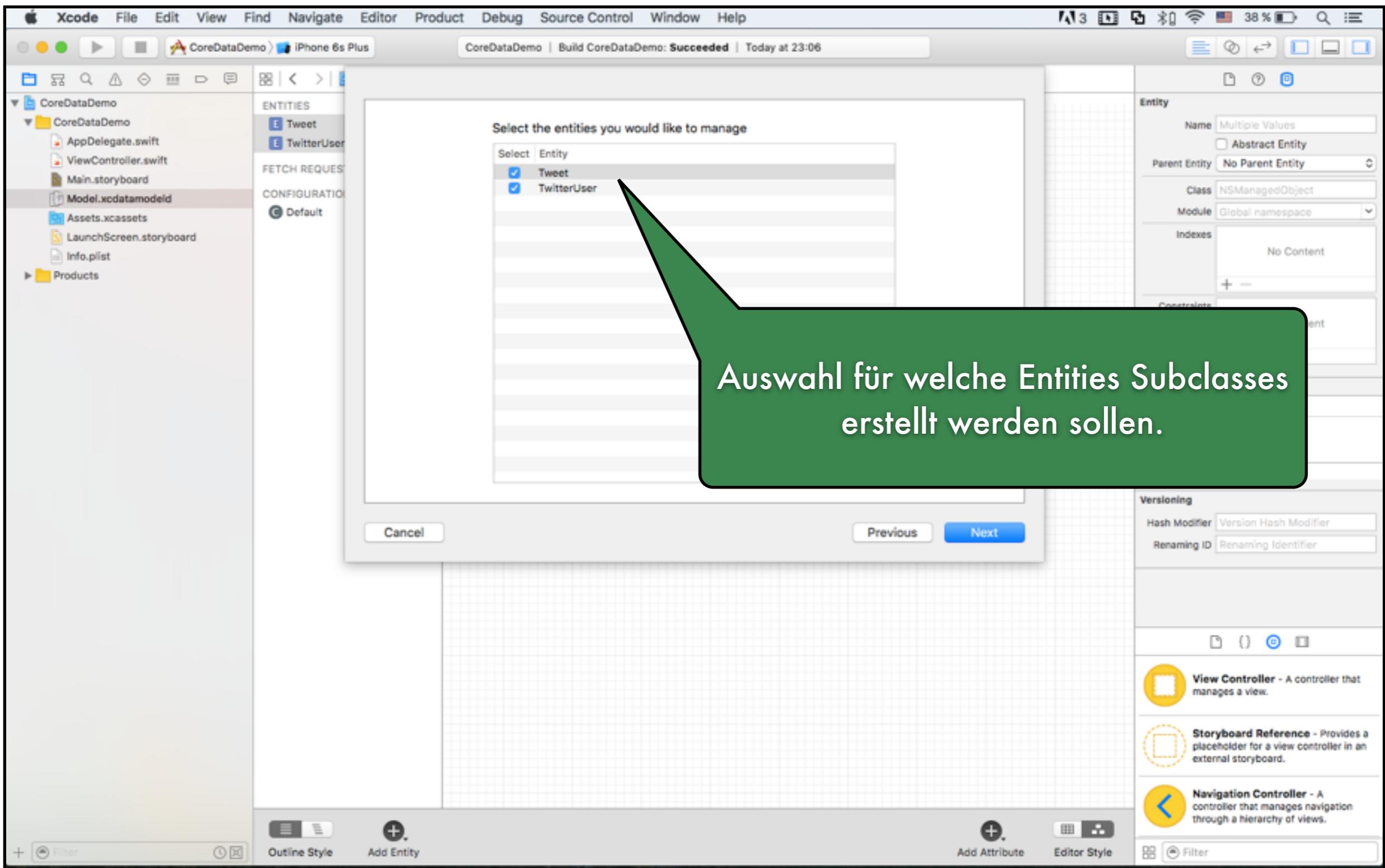
# Core Data



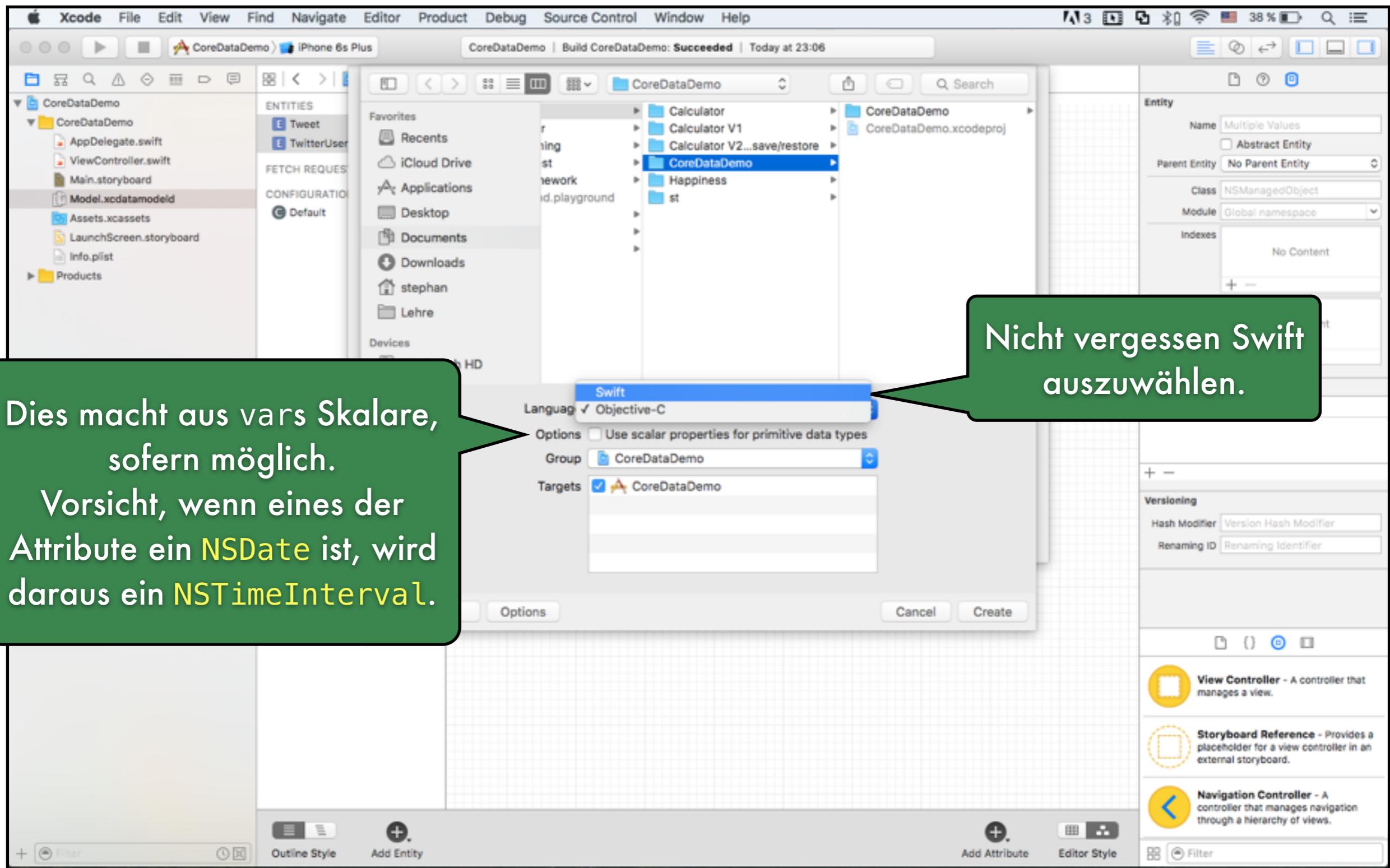
# Core Data



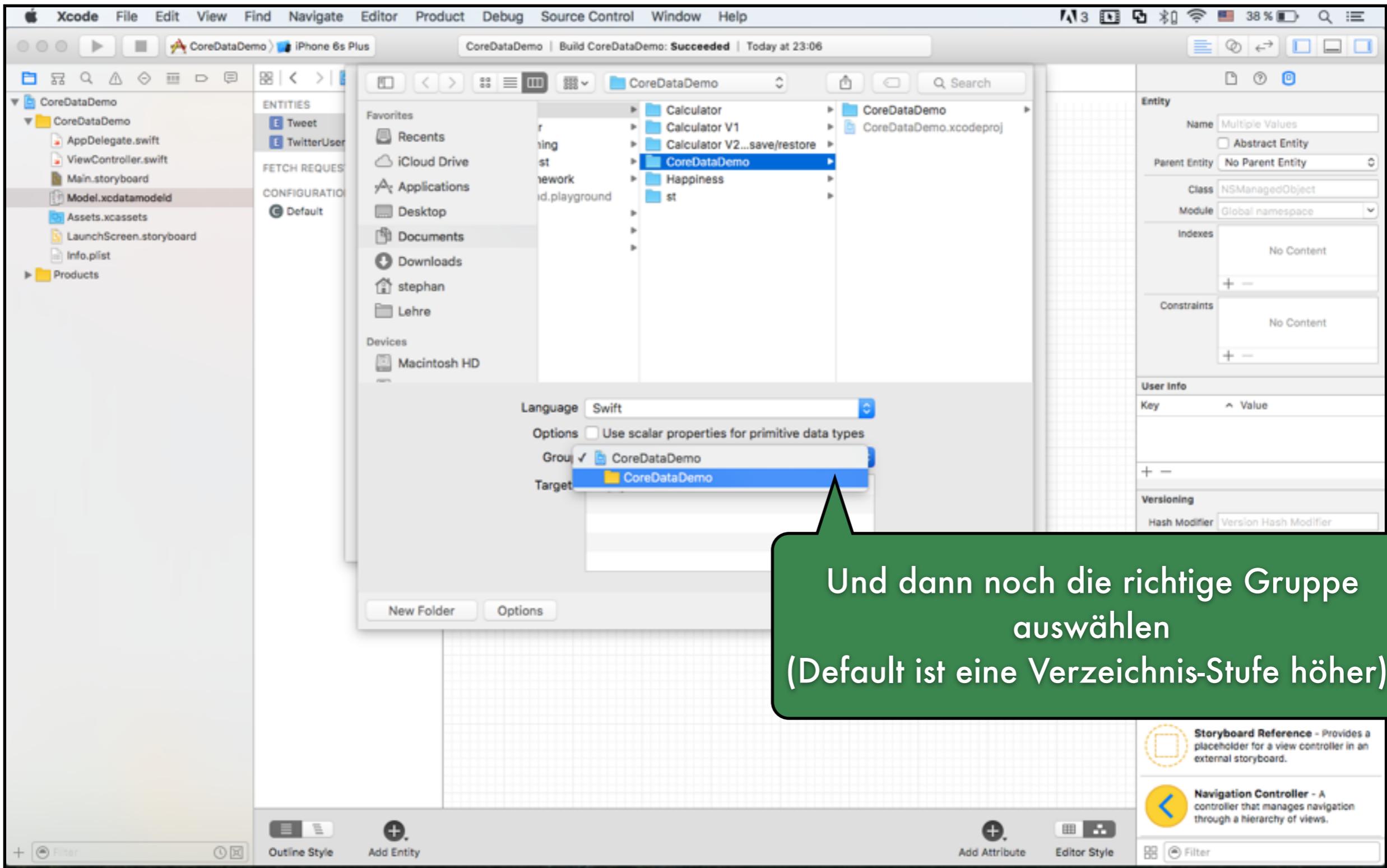
# Core Data



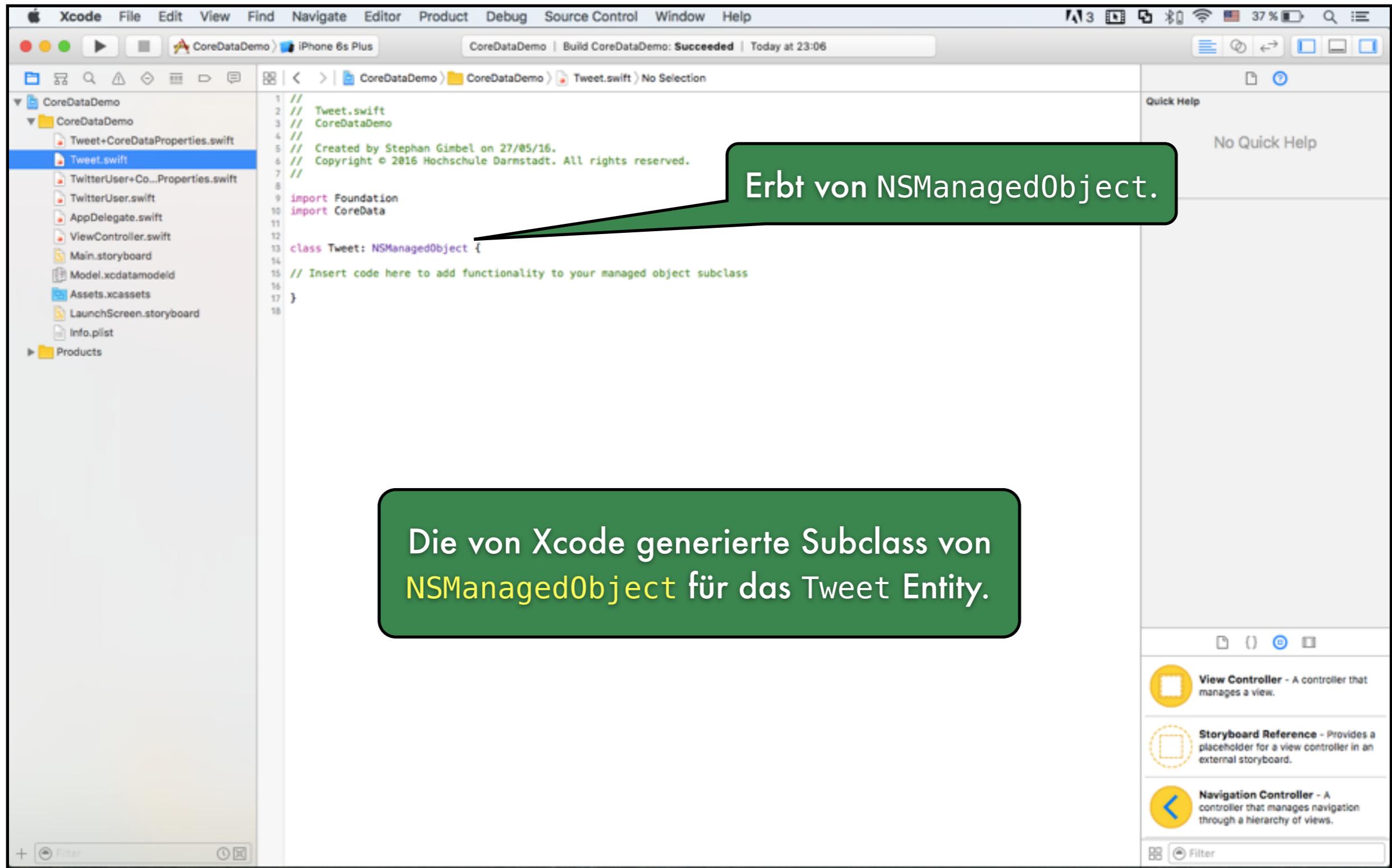
# Core Data



# Core Data



# Core Data



The screenshot shows the Xcode interface with the project 'CoreDataDemo' open. The 'Tweet.swift' file is selected in the left sidebar. The code editor displays the following Swift code:

```
//  
//  Tweet.swift  
//  CoreDataDemo  
//  
//  Created by Stephan Gimbel on 27/05/16.  
//  Copyright © 2016 Hochschule Darmstadt. All rights reserved.  
  
import Foundation  
import CoreData  
  
class Tweet: NSManagedObject {  
    // Insert code here to add functionality to your managed object subclass  
}
```

A green callout bubble points to the line 'class Tweet: NSManagedObject {'. It contains the text 'Erbt von NSManagedObject.'.

A green callout bubble points to the entire class definition 'class Tweet: NSManagedObject { ... }'. It contains the text 'Die von Xcode generierte Subclass von NSManagedObject für das Tweet Entity.'

The Xcode interface includes a toolbar at the top, a status bar at the bottom right, and a navigation bar with the project name 'CoreDataDemo' and build status 'Build CoreDataDemo: Succeeded | Today at 23:06'.

# Core Data

The screenshot shows the Xcode interface with the following details:

- File menu:** Xcode, File, Edit, View, Find, Navigate, Editor, Product, Debug, Source Control, Window, Help.
- Toolbar:** Standard Xcode toolbar with icons for play, stop, run, etc.
- Project Navigator:** Shows the project structure under "CoreDataDemo".
- Editor:** Displays the code for `TwitterUser.swift`. The code is as follows:

```
// TwitterUser.swift
// CoreDataDemo
//
// Created by Stephan Gimbel on 27/05/16.
// Copyright © 2016 Hochschule Darmstadt. All rights reserved.

import Foundation
import CoreData

class TwitterUser: NSManagedObject {

    // Insert code here to add functionality to your managed object subclass
}


```

A callout bubble in the center of the screen contains the text: "... und eine weitere Subclass für das TwitterUser Entity."

The bottom right corner of the Xcode interface shows a sidebar with three items:

- View Controller** - A controller that manages a view.
- Storyboard Reference** - Provides a placeholder for a view controller in an external storyboard.
- Navigation Controller** - A controller that manages navigation through a hierarchy of views.

# Core Data

Xcode File Edit View Find Navigate Editor Product Debug Source Control Window Help

CoreDataDemo | Build CoreDataDemo: Succeeded | Today at 23:06

CoreDataDemo CoreDataDemo CoreDataDemo Tweet+CoreDataProperties.swift No Selection

Quick Help  
No Quick Help  
Search Documentation

CoreDataDemo

Tweet+CoreDataProperties.swift

Tweet.swift

TwitterUser+CoreDataProperties.swift

TwitterUser.swift

AppDelegate.swift

ViewController.swift

Main.storyboard

Model.xcdatamodeld

Assets.xcassets

LaunchScreen.storyboard

Info.plist

Products

```
1 // Copyright © 2016 Stephan Gimbel. All rights reserved.
2 // Choose "Create NSManagedObject Subclass..." from the Core Data editor menu
3 // to delete and recreate this implementation file for your updated model.
4 //
5
6 import Foundation
7 import CoreData
8
9 extension Tweet {
10     @NSManaged var text: String?
11     @NSManaged var created: NSDate?
12     @NSManaged var id: String?
13     @NSManaged var tweeter: TwitterUser?
14 }
15
16
17
18
19
20
21
22
23 }
```

Welches File wurde noch generiert?

Die Typen werden ebenfalls passend generiert (Reihenfolge beachten).

Es ist eine Extension der Tweet Klasse, welche es uns erlaubt mittels vars auf alle Attribute zuzugreifen.

View Controller - A controller that manages a view.

Storyboard Reference - Provides a placeholder for a view controller in an external storyboard.

Navigation Controller - A controller that manages navigation through a hierarchy of views.

# Core Data

The screenshot shows the Xcode interface with a project named "CoreDataDemo". The left sidebar shows files like Tweet+CoreDataProperties.swift, Tweet.swift, and TwitterUser+CoreDataProperties.swift. The main editor window displays Swift code for generating NSManagedObject subclasses:

```
// TwitterUser+CoreDataProperties.swift
// CoreDataDemo
//
// Created by Stephan Gimbel on 27/05/16.
// Copyright © 2016 Hochschule Darmstadt. All rights reserved.
//
// Choose "Create NSManagedObject Subclass..." from the Core Data editor menu
// to delete and recreate this implementation file for your updated model.
//
import Foundation
import CoreData

extension TwitterUser {

    @NSManaged var screenName: String?
    @NSManaged var name: String?
    @NSManaged var tweets: NSSet?
}


```

Three callout bubbles highlight specific features:

- A green callout bubble points to the `@NSManaged` annotations in the code, stating: "@NSManaged sagt Swift nur, dass die NSManagedObject Subclass diese Properties speziell behandelt (es macht für uns valueForKey/setValue(forKey:))".
- A green callout bubble points to the generated `extension TwitterUser { ... }` block, stating: "... ebenfalls für TwitterUser."
- A green callout bubble points to the generated `var tweets: NSSet?` property, stating: "Die Typen werden ebenfalls passend generiert (Reihenfolge beachten)."

The Xcode interface includes a toolbar, a status bar at the top, and a navigation bar at the bottom.

# Core Data

---

Wie können wir nun auf die Attribute des Entities in dieser Subclass zugreifen?

Erstellen eines Tweet Entity in der Datenbank...

```
let context = document.managedObjectContext // oder aus AppDelegate  
if let tweet = NSEntityDescription.insertNewObjectForEntityForName("Tweet",  
                           inManagedObjectContext:context) as? Tweet {  
    tweet.text = "Some really important thing I need to share"  
    tweet.created = NSDate()  
    tweet.tweeter = ... // some TwitterUser  
    tweet.tweeter.name = "Tim Cook" // with chaining  
}
```

Dies ist deutlich eleganter als

setValue("Some really important thing I need to share", for Key: "text")

Und Swift kann Type Checking durchführen.

# Core Data

---

## Löschen

Das Löschen von Objekten aus der Datenbank ist einfach (manchmal zu einfach!)

```
managedObjectContext.deleteObject(tweet)
```

Wir müssen aber sicherstellen dass die restlichen Objekte in der Datenbank nach dem Löschen in einem sinnvollen Zustand sind.

Relationships werden für uns geupdated (wenn die Delete Rule für Relationship Attribute korrekt gesetzt sind).

Keine strong Pointer zu `tweet` halten, nachdem es gelöscht wurde.

## prepareForDeletion

Diese Methode können wir in der NSManagedObject Subclass implementieren...

```
func prepareForDeletion() {  
    // Wir müssen hier nicht den tweeter zu nil setzen (das geschieht  
    // automatisch).  
    // aber wenn der TwitterUser z.B. ein "number of tweets tweeted"  
    // Attribut hat, dann könnten wir hier die Anzahl anpassen  
    // (tweeter.tweetCount -= 1).  
}
```

# Core Data

---

Bisher können wir...

Objekte in der Datenbank erstellen:

`insertNewObjectForEntityForName(inManagedObjectContext:)`

Get/Set von Properties mit `valueForKey/setValue(forKey:)` oder vars in einer custom Subclass.

Objekte löschen mit der `NSManagedObjectContext` Methode `deleteObject`.

Eine weitere wichtige Funktion ist die Abfrage (Query)

Wir wollen auch in der Lage sein Objekte aus der Datenbank zu erhalten, sie nicht nur neu zu erstellen.

Dies machen wir indem wir ein `NSFetchRequest` in unserem `NSManagedObjectContext` ausführen.

Vier wichtige Dinge, um ein `NSFetchRequest` zu erstellen

1. Entity welches wir fetchen wollen (required)
2. Wie viele Objekte wir auf einmal fetchen wollen und/oder das Maximum zum fetchen (optional, default: all)
3. NSSortDescriptors um die Reihenfolge des Arrays zu spezifizieren, welches für die gefetchten Objekte zurückgegeben wird
4. NSPredicate um zu spezifizieren welche dieser Entities gefetched werden (optional, default: all)

# Core Data

---

## Erstellen eines NSFetchedRequest

Wir betrachten dies Zeile für Zeile...

```
let request = NSFetchedRequest(entityName: "Tweet")
request.fetchBatchSize = 20
request.fetchLimit = 100
request.sortDescriptors = [sortDescriptor]
request.predicate = ...
```

## Welche Art con Entity wollen wir fetchen

Ein fetch gibt Objekte der selben Art von Entity zurück.

Wir können keine Fetches haben welche einige Tweets und einige TwitterUser zurückgeben (entweder das eine oder das andere).

## Setzen der Fetch Size/Limits

Wenn wir einen Fetch erstellen der 1000 Objekte matched, dann würde das Fetch oben 20 Stück gleichzeitig erhalten (Faulting).

Und würde ebenfalls stoppen nachdem 100 von den 1000 gefetched wurden.

# Core Data

---

## NSSortDescriptor

Wenn wir ein Fetch Request ausführen, gibt es ein **Array** von NSManagedObjects zurück.

Arrays sind "geordnet", also sollten wir auch die Reihenfolge angeben wenn wir **fetchen**.

Dies machen wir, indem wir dem Fetch Request eine Liste von "Sort Descriptors" übergeben, die beschreiben wie sortiert werden soll.

```
let sortDescriptor = NSSortDescriptor(  
    key: screenName",  
    ascending: true,  
    selector: #selector(NSString.localizedStandardCompare(_)))  
)
```

Der **selector:** Parameter ist nur eine Methode, die an jedes Objekt geschickt wird um es mit einem anderen zu vergleichen.

Einige dieser "Methoden" sind vielleicht "smart" (d.h. sie können auf der Datenbank Seite ausgeführt werden).

Wir geben ein Array dieser NSSortDescriptors an NSFetchedRequest, da wir manchmal zuerst nach einem Key (z.B. Nachname) und dann, innerhalb dieser Sortierung, nach einem weiteren Kriterium sortieren wollen (z.B. Vorname).

Beispiel: [lastNameSortDescriptor, firstNameSortDescriptor]

# Core Data

---

## NSPredicate

Hier spezifizieren wir ganz genau welche Objekte wir aus der Datenbank haben wollen.

## Predicate Formate

Wir erstellen dies mit einem formatierten String mit starker semantischer Bedeutung (siehe NSPredicate Dokumentation).

Beachten, dass wir `%@` (ähnlich `printf`) statt `\(expression)` verwenden um Daten zu spezifizieren.

```
let searchString = "foo"
let predicate = NSPredicate(format: "text contains[c] %@", searchString)
let tim: TwitterUser = ... // a TwitterUser
let predicate = NSPredicate(format: "tweeter = %@ && created > %@", tim, aDate)
let predicate = NSPredicate(format: "tweeter.screenName = %@", "apple")
```

Dies sind nur Predicates für Suchen im Tweet Table.

Hier ein Predicate welches eine Suche nach TwitterUser durchführt...

```
let predicate = NSPredicate(format: "tweets.text contains %@", searchString)
```

Dies würde einen TwitterUser (nicht Tweet) finden, dessen Tweets den String enthalten.

# Core Data

---

## NSCompoundPredicate

Wir können AND und OR in einem predicate String verwenden,

z.B. @“(name = %@) OR (title = %@)”

Oder wir können NSPredicate Objekte mit speziellen NSCompoundPredicates kombinieren.

```
let array = [predicate1, predicate2]
```

```
let predicate = NSCompoundPredicate(andPredicateWithSubpredicates: array)
```

Dieses Predicate ist “predicate1 AND predicate2”.

# Core Data

---

## Fortgeschrittene Query - Key Value Coding

Kann Predicates wie “`tweets.@count > 5`” (TwitterUser mit mehr als 5 Tweets)

`@count` ist eine Funktion (es gibt weitere) die in der Datenbank selbst ausgeführt wird.

<https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/KeyValueCoding/Articles/CollectionOperators.html>

All dies funktioniert ebenfalls für Dictionarys, Arrays und Sets ... (und noch mehr)

z.B. `propertyList.valueForKeyPath("tweets.tweet.@avg.latitude")`

gibt durchschnittlichen Breitengrad aller Tweets zurück (ja, wirklich!)

z.B. “`tweets.tweet.text.length`” gibt ein Array mit der Länge der Texte der Tweets zurück.

## NSExpression

Fortgeschrittenes Thema. Kann komplexe Daten aus der Datenbank sammeln.

Wir haben leider keine Zeit dies zu behandeln.

Bei Interesse, für NSExpression und Key Value Coding mal folgende Queries untersuchen...

```
let request = NSFetchedRequest("...")  
request.resultType = .DictionaryResultType // Fetch gibt Arrays von  
// Dictionary statt NSM0s zurück  
request.propertiesToFetch = ["name", expression, etc.]
```

# Core Data

---

## Zusammenfassung

**Angenommen wir wollen nach allen TwitterUsern suchen...**

```
let request = NSFetchedRequest(entityName: "TwitterUser")
```

**... die einen Tweet in den letzten 24 Stunden erstellt haben...**

```
let yesterday = NSDate(timeIntervalSinceNow: -24*60*60)
```

```
request.predicate = NSPredicate(format: "any tweets.created > %@", yesterday)
```

**... sortiert nach TwitterUsers Name ...**

```
request.sortDescriptors = [NSSortDescriptor(key: "name", ascending: true)]
```

# Core Data

---

## Ausführen des Fetch

```
context = aDocument.managedObjectContext // oder AppDelegate var  
let users = try? context.executeFetchRequest(request)
```

Wir haben hier eine andere Art von `try?` verwendet.

Das `?` bedeutet "versuche dies und wenn es einen error wirft, gib mir nil zurück".

Wir könnten auch normal `try` verwenden zusammen mit `do { } und errors catchen`, wenn wir daran interessiert sind.

Ansonsten gibt die ausführende Fetch Request Methode zurück...

ein leeres Array (nicht nil) wenn es erfolgreich ist und keine matches in der Datenbank sind.

ein `Array` von `NSManagedObjects` (oder Subclass davon) wenn matches existieren.

That's it. As simple as that.

# Core Data

---

## Faulting

Diese Fetches empfangen nicht notwendigerweise Daten.

Es kann auch ein Array von "unfaulted" Objekten sein, welche aus den Zugriff warten.

Core Data ist "smart" bzgl. Faulting der Daten und den Zugriff darauf.

Wenn wir z.B. dies versuchen...

```
for user in twitterUser) {  
    print("fetched user \(user)")  
}
```

Vielleicht gibt dies die Namen der User aus, vielleicht sehen wir aber auch nur "unfaulted" Objekte. Dies ist abhängig davon ob prefetched wurde.

Wenn wir aber dies machen...

```
for user in twitterUsers) {  
    print("fetched user named \(user.name)")  
}
```

Dann würden wir definitiv alle TwitterUser aus der Datenbank faulten.

Dies liegt daran, dass wir hier tatsächlich auf die Daten der NSManagedObjects zugreifen.

# Core Data

---

**NSManagedObjectContext ist nicht Thread Safe**

Core Data Zugriff ist sehr schnell, daher wird Multithreading oft nicht benötigt.

NSManagedObjectContexts werden unter Verwendung eines "queue-based concurrency Models" erstellt.

Dies bedeutet, wird können auf einen Context und seine NSMOs nur in der Queue zugreifen, in der sie erstellt wurden.

Queue bedeutet hier eine "serial Queue", nicht die QoS-basierten Queues.

Die am meisten verwendete Queue ist die Main Queue (UIManagedDocument oder AppDelegate).

Es ist möglich eigene NSManagedObjectContexts in anderen seriellen Queues zu erstellen.

**Thread Safer Zugriff auf NSManagedObjectContext**

```
context.performBlock { // oder performBlockAndWait bis beendet  
    // bearbeiten von Context in der safe queue (die queue in der er  
    // erstellt wurde)  
}
```

Beachten dass die Queue durchaus die Main Queue sein kann, also nicht notwendigerweise multithreaded.

Es ist aber eine gute Idee alle Calls zu einem NSManagedObjectContext so durchzuführen.

Wenn es kein Multithreading ist, ist dies nicht "teurer".

# Core Data

---

## Parent Context

Einige Contexts (inkl. `UIManagedDocument`) haben einen `parentContext` (var des NSMOC).

Dieser `parentContext` ist meist in einer separaten Queue, greift aber auf die gleiche Datenbank zu.

Dies bedeutet, dass `performBlock` darauf ausgeführt werden kann, um auf die Datenbank außerhalb der Main Queue zuzugreifen.

Aber es ist immer noch ein anderer Context, also muss ein Refetch im child durchgeführt werden um Änderungen zu sehen.

Es gibt noch viel, viel mehr

Leider haben wir nicht die Zeit uns dies alles anzuschauen...

Optimistic Locking (`deleteConflictsForObject`)

Rollback von unsaved Changes

Undo/Redo

Staleness (wie lange nach einem Fetch bis ein Objekt ein Refetch benötigt?)

...

# Core Data

---

## NSFetchedResultsController

**Verbindet einen NSFetchedRequest mit einem UITableViewController.**

**Normalerweise haben wir eine NSFetchedResultsController var in unserem UITableViewController.**

**Wird mit einem NSFetchedRequest verbunden der alle Daten zurück gibt, die angezeigt werden sollen.**

**Dann kann der NSFRC verwendet werden um alle Anforderungen an das UITableViewDataSource Protocol zu erfüllen.**

## Beispiel

```
var fetchedResultsController = ...  
func numberOfSectionsInTableView(sender: UITableView) -> Int {  
    return fetchedResultsController?.sections?.count ?? 1  
}  
func tableView(sender: UITableView, numberOfRowsInSection section: Int) -> Int {  
    if let sections = fetchedResultsController?.sections where sections.count > 0 {  
        return sections[section].numberOfObjects  
    } else {  
        return 0  
    }  
}
```

# Core Data

---

**Wichtige Methode objectAtIndexPath**

**NSFetchedResultsController Methode...**

func objectAtIndexPath(indexPath: NSIndexPath) -> NSManagedObject

**Können wir so z.B. in tableView(cellForRowAtIndexPath:) verwenden...**

```
func tableView(tv: UITableView, cellForRowAt indexPath: NSIndexPath) -> UITableViewCell {  
    let cell = tv.dequeueReusableCell(...)  
    if let obj = fetchedResultsController.objectAtIndexPath(indexPath) as? Tweet {  
        // cell füllen mit den properties von objs  
    }  
    return cell  
}
```

# Core Data

---

## Erstellen eines NSFetchedResultsController

Benötigt ein NSFetchedResultsController um zu funktionieren (und ein NSManagedObjectContext aus dem gefetched werden kann).

Wenn wir alle tweets von jemandem mit dem Namen theName empfangen und in unserem Table anzeigen wollen...

```
let request = NSFetchedResultsController(entityName: "Tweet")
request.sortDescriptors = [NSSortDescriptor(key: "created" ...)]
request.predicate = NSPredicate(format: "tweeter.name = %@", theName)

let frc = NSFetchedResultsController(
    fetchRequest: request,
    managedObjectContext: context,
    sectionNameKeyPath: keyThatSaysWhichAttributesIsTheSectionName,
    cacheName: "MyTwitterQueryCache")
```

Sicherstellen dass der `cacheName` immer mit genau dem selben `request` assoziiert ist.

Es ist ok hier `nil` anzugeben für den `cacheName`, dann findet kein Caching für die Results statt.

Es ist wichtig, dass der `sortDescriptor` mit `keyThatSaysWhichAttribute...` übereinstimmt. Die Results müssen so sortiert sein, dass alle Objekte in der ersten Section zuerst kommen, die zweiten als zweites, usw.

# Core Data

NSFRC "beobachtet" Änderungen in Core Date und kann Tables auto-updaten  
Verwendet Key-Value Observing Mechanismus.

Wenn eine Änderung bemerkt wird, schickt es eine Message zu seinem delegate...

```
func controller ( NSFetchedResultsController,  
                  didChangeObject: AnyObject  
                  atIndexPath: NSIndexPath?  
                  forChangeType: NSFetchedResultsChangeType  
                  newIndexPath: NSIndexPath?)  
{  
    // hier die passenden UITableView Methoden aufrufen um die rows zu updaten  
}
```

NSFetchedResultsController Doku zeigt wie all dies funktioniert

Tatsächlich langt es aus den Code aus der Doku in die eigene TableView Subclass zu kopieren.