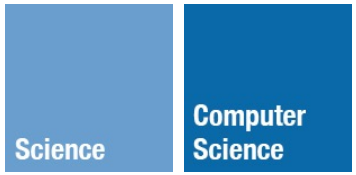
Core Data



CS 442: Mobile App Development

Michael Saelee <lee@iit.edu>

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

persistence framework

(not just an ORM, as non-relational backends are supported)

CD tracks an *object graph* (possibly disjoint), and manages its persistence automatically

— supports on-demand loading & flushing

Does this via an explicitly referenced *context* — *all CD-managed objects* must be created in this context to be automatically persisted

§ Core Data class overview

CD-managed objects are of type `NSManagedObject`

= model instance / “entity”

≈ row in database table (for ORM)

- managed object may be a “fault” until data is needed
- when fault “fires”, object is hydrated from persistent store

how does Core Data know what entities we have, and what properties/relationships are defined for each entity?

`NSManagedObjectModel`

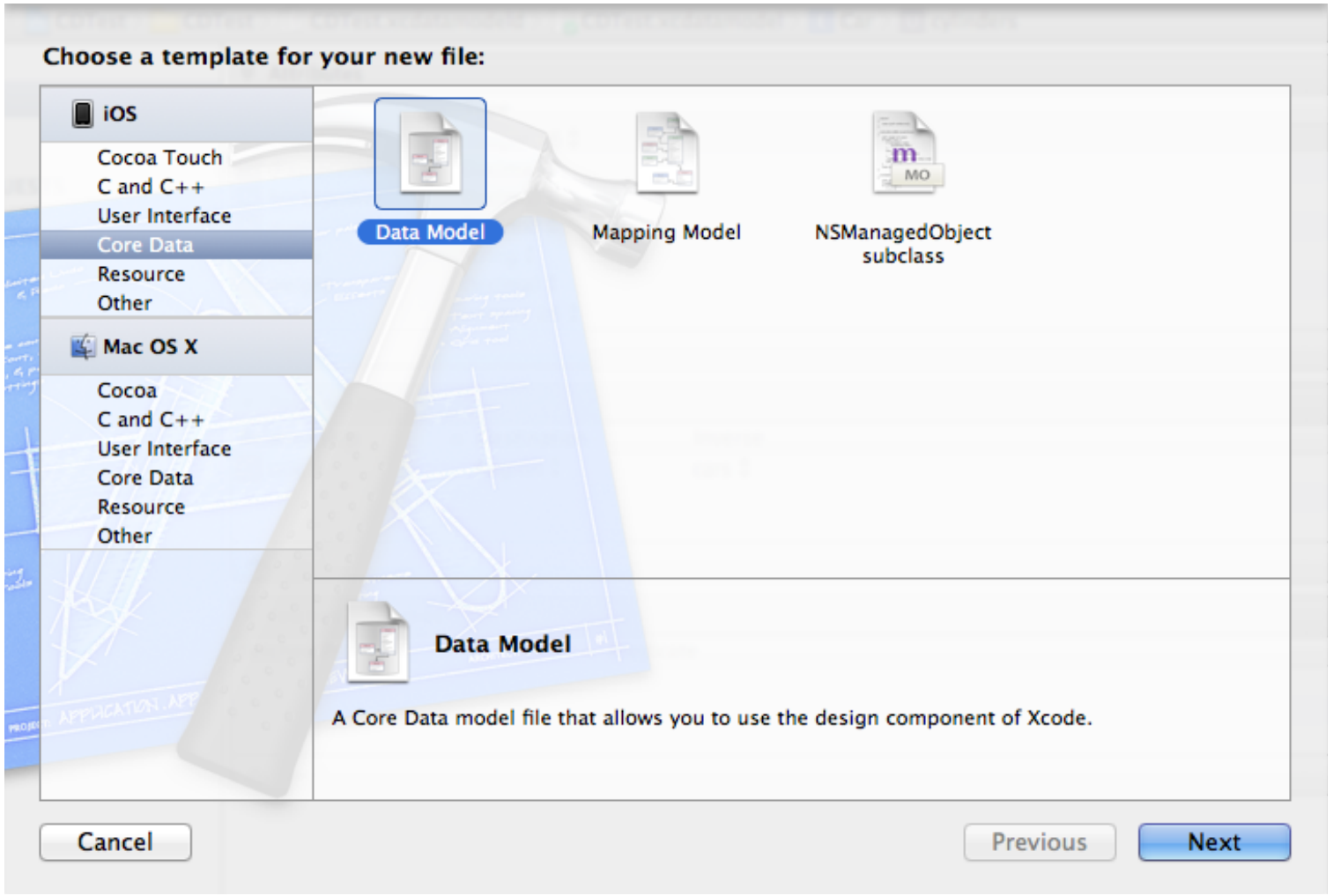
is used to describe our models/entities

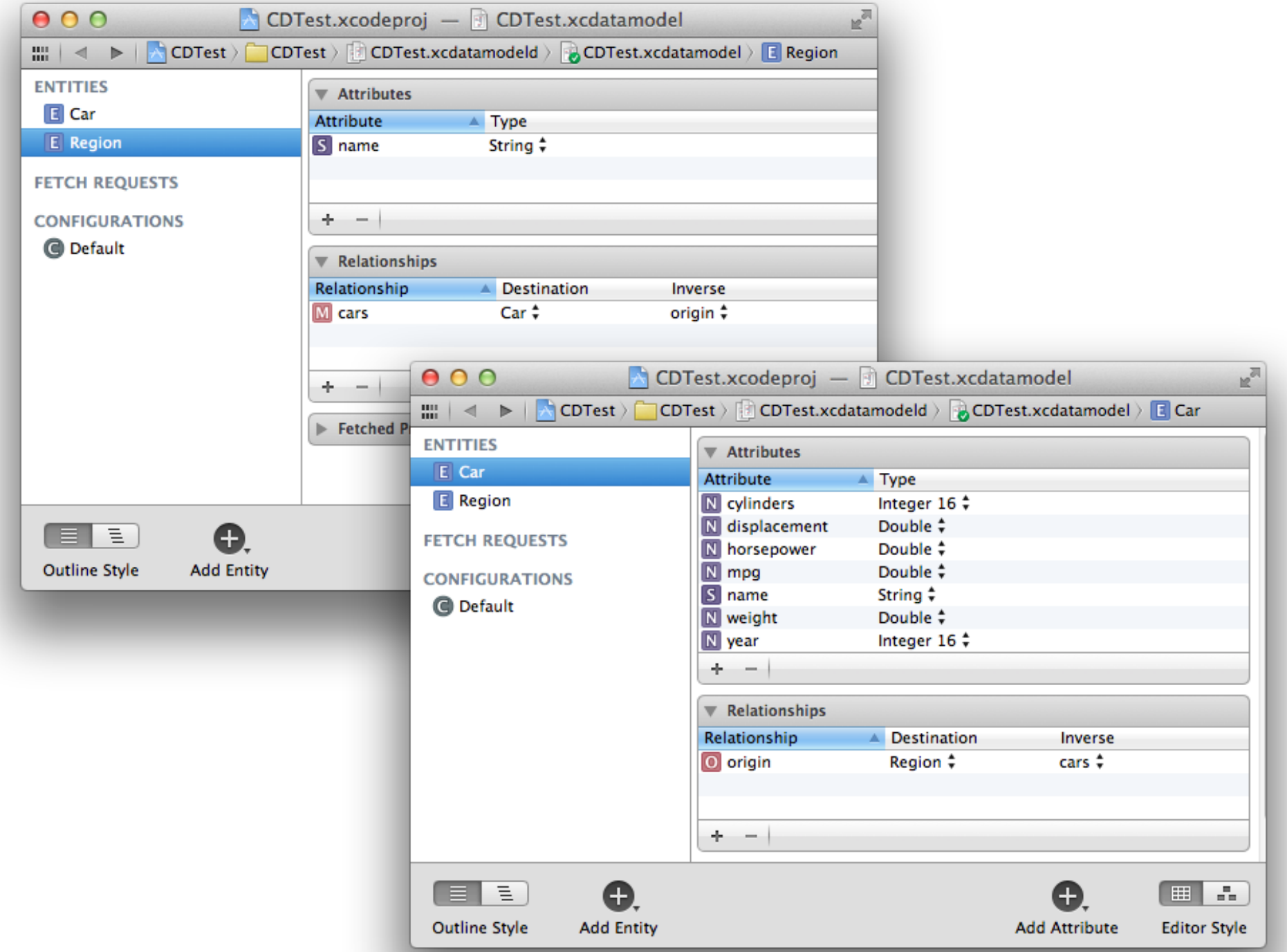
≈ database schema

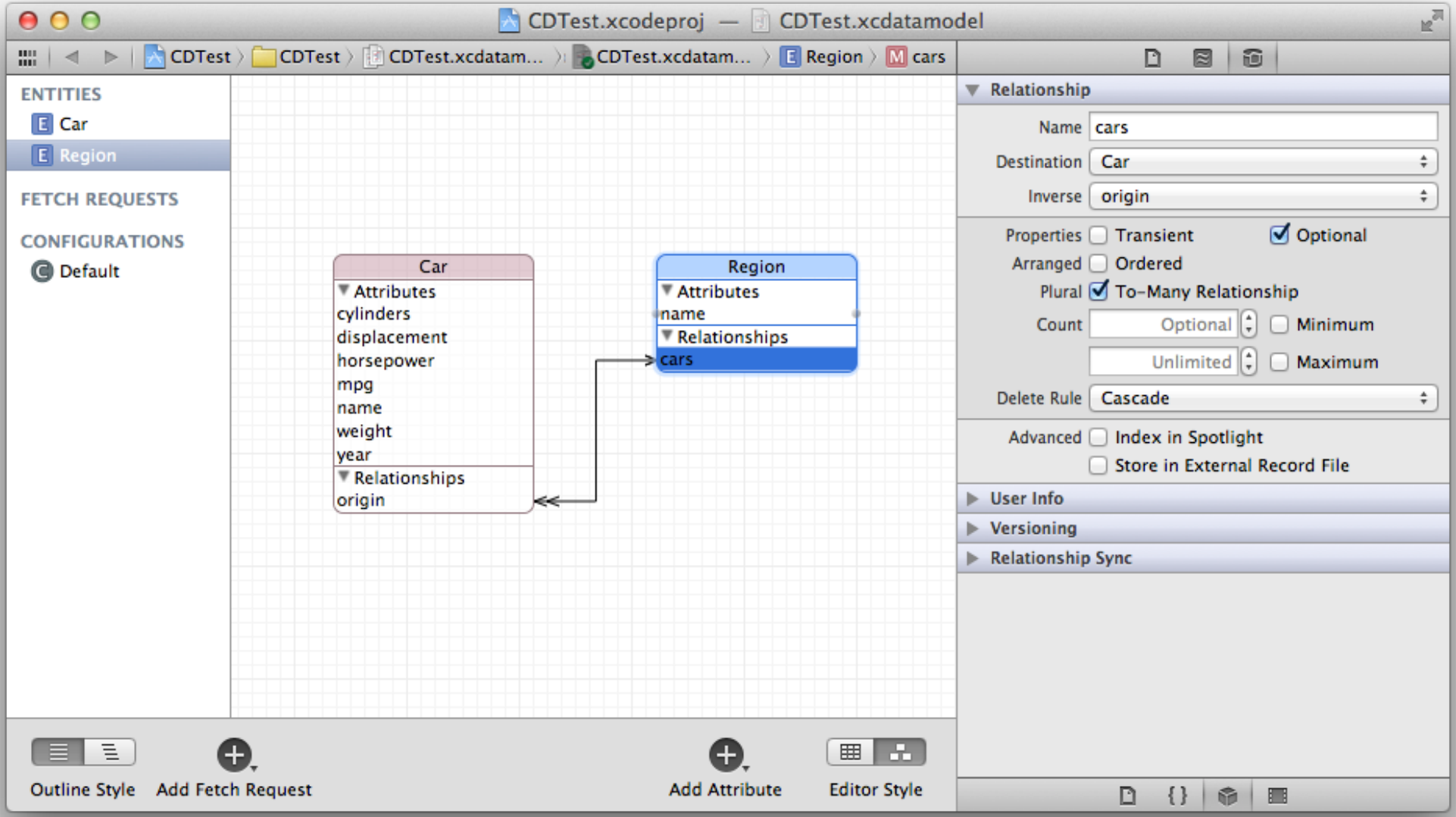
`NSManagedObjectModel`
is a collection of per-entity
`NSEntityDescription` objects

≈ table definition

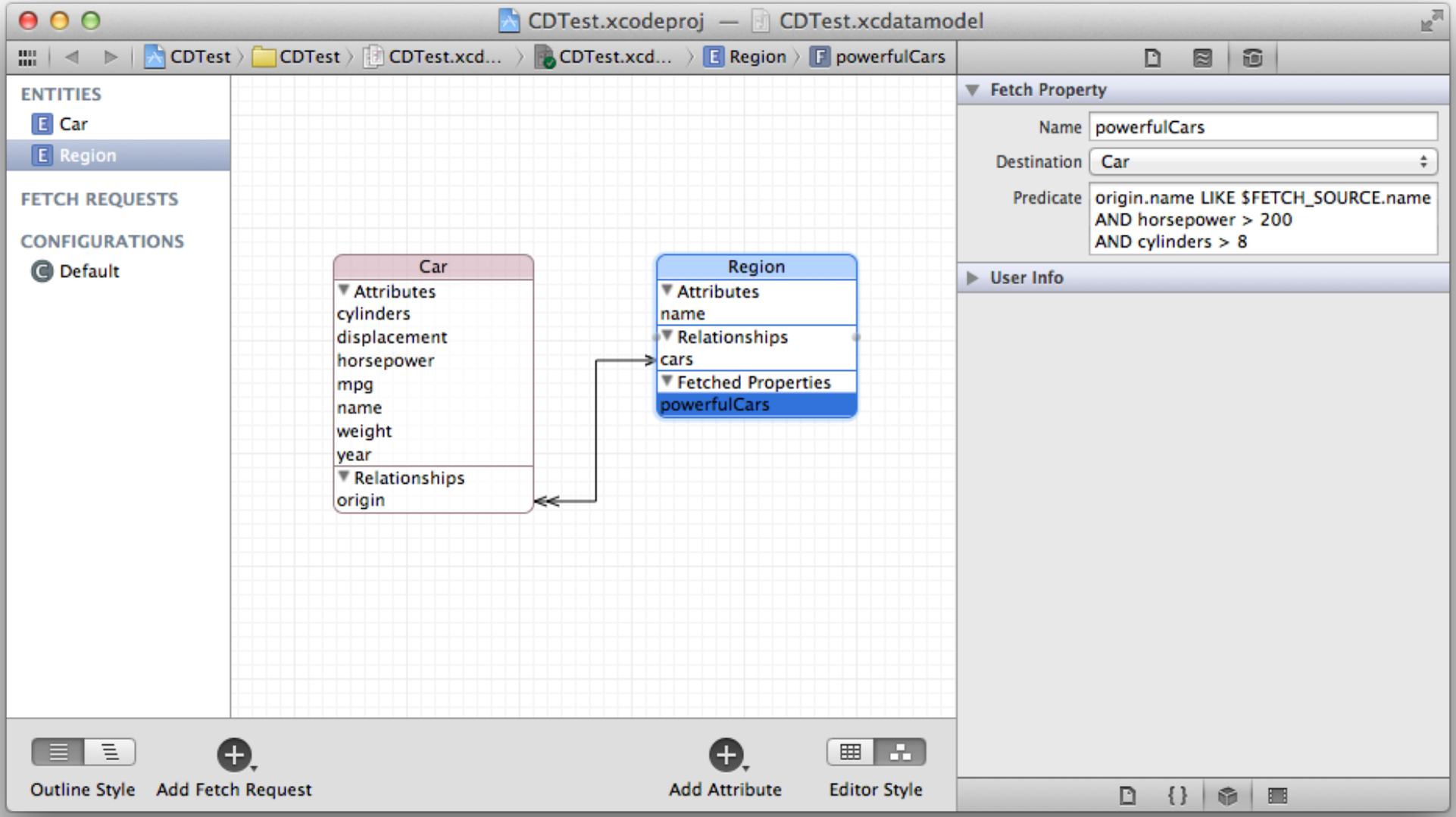
can be created programmatically, but *painful!*


Choose a template for your new file:
iOS
Cocoa Touch
C and C++
User Interface
Core Data
Resource
Other
Mac OS X
Cocoa
C and C++
User Interface
Core Data
Resource
Other
Data Model
Mapping Model
MO
NSManagedObject subclass
Data Model
A Core Data model file that allows you to use the design component of Xcode.
Cancel
Previous
Next


CDTest.xcodeproj — CDTest.xcdatamodel
CDTest › CDTest › CDTest.xcdatamodeld › CDTest.xcdatamodel › Region
ENTITIES
Car
Region
FETCH REQUESTS
CONFIGURATIONS
Default
Attributes
Attribute
Type
name
String
Relationships
Relationship
Destination
Inverse
cars
Car
origin
Outline Style
Add Entity
CDTest.xcodeproj — CDTest.xcdatamodel
CDTest › CDTest › CDTest.xcdatamodeld › CDTest.xcdatamodel › Car
ENTITIES
Car
Region
FETCH REQUESTS
CONFIGURATIONS
Default
Attributes
Attribute
Type
cylinders
Integer 16
displacement
Double
horsepower
Double
mpg
Double
name
String
weight
Double
year
Integer 16
Relationships
Relationship
Destination
Inverse
origin
Region
cars
Outline Style
Add Entity
Add Attribute
Editor Style



entity relationships



transient “fetched properties”

compiles to “.mom” file

- typically one .mom per app
- merging of multiple .moms automatically done (if needed)

```
let modelURL = NSBundle.mainBundle().URLForResource("Cars", withExtension: "momd")!
let mom = NSManagedObjectModel(contentsOfURL: modelURL)
for entity in mom!.entities as [NSEntityDescription] {
  println(entity.name!)
  for property in entity.properties as [NSPropertyDescription] {
    print("- \(property.name)")
    switch property {
    case let attribute as NSAttributeDescription:
      println(": \(attribute.attributeValueClassName!)")
    case let relationship as NSRelationshipDescription:
      println(" -> \(relationship.destinationEntity!.name!)")
   }
 }
}
```

```
Cars[43101:207] Car
Cars[43101:207] - cylinders: NSNumber
Cars[43101:207] - displacement: NSDecimalNumber
Cars[43101:207] - horsepower: NSDecimalNumber
Cars[43101:207] - mpg: NSDecimalNumber
Cars[43101:207] - name: NSString
Cars[43101:207] - weight: NSDecimalNumber
Cars[43101:207] - year: NSNumber
Cars[43101:207] - origin -> Region
Cars[43101:207] Region
Cars[43101:207] - name: NSString
Cars[43101:207] - cars -> Car
```

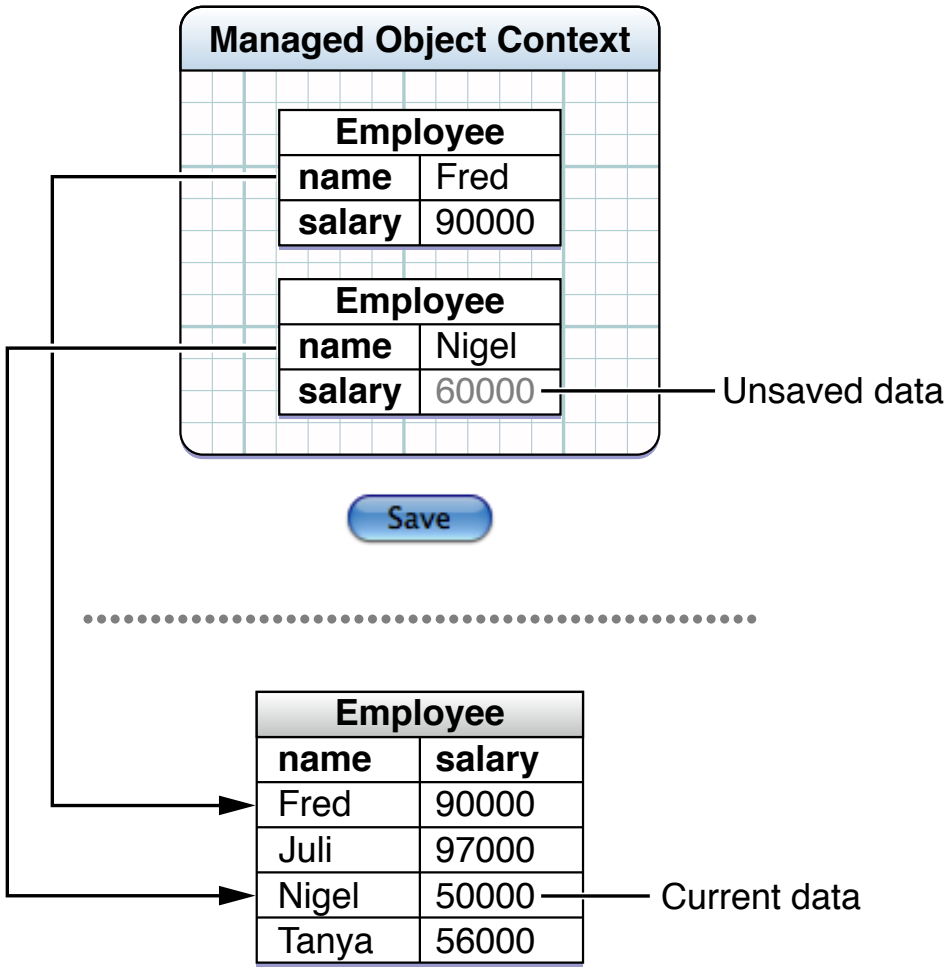
recall: managed objects must be explicitly associated with a CD *context*

`NSManagedObjectContext`

- *all managed objects* live in one of these
  - a given managed object lives in *one and only one* MOC
    - MOC guarantees object uniqueness

MOC acts as a “scratchpad” for new/updated managed objects

— explicit save required for persistence


Managed Object Context
Employee
name
Fred
salary
90000
Employee
name
Nigel
salary
60000
Unsaved data
Save
Employee
name
salary
Fred
90000
Juli
97000
Nigel
50000
Current data
Tanya
56000

```swift
func applicationWillTerminate(application: UIApplication) {
  self.saveContext()
}


func saveContext () {
  if let moc = self.managedObjectContext {
    var error: NSError? = nil
    if moc.hasChanges && !moc.save(&error) {
     NSLog("Unresolved error \(error), \(error!.userInfo)")
      abort()
    }
  }
}
```

MOC also buys us free undo management

```
managedObjectContext.undoManager = NSUndoManager()
```

Undo Management

- undo
- redo
- rollback
- save:
- hasChanges

note that a MOC is ***not*** thread-safe!

(not really a good idea anyway, to share a huge cache of mutable objects between threads)

MOC is *backing store agnostic*

i.e., it doesn't matter what sort of storage mechanism is actually used for persistence

`NSPersistentStoreCoordinator`

wraps and manages the backing store

— almost never use use directly

supported persistent/disk stores are **XML**, **binary**, and **SQLite**; **memory-based** store is also available

SQLite is the only persistent store type that supports lazy/partial loading (*best scalability!*)

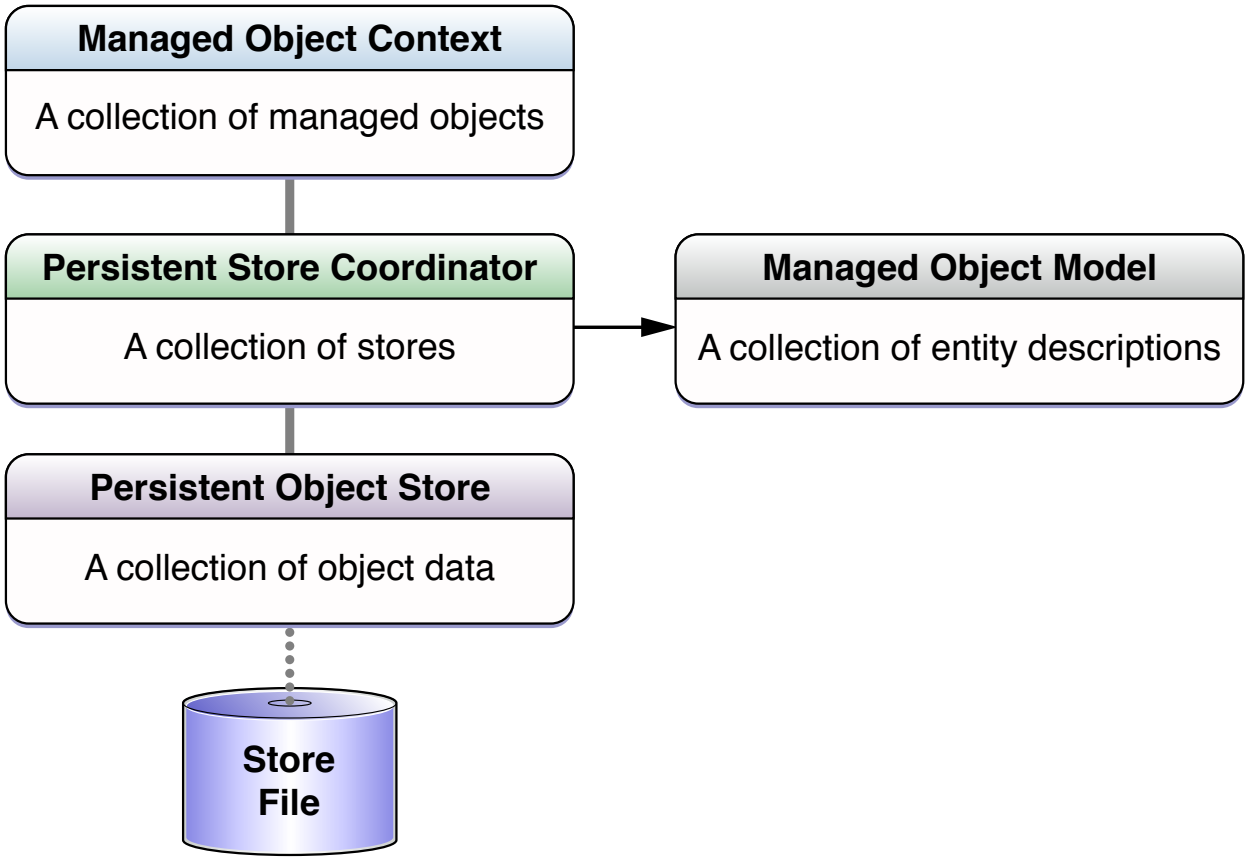
¶ Recap:

NSManagedObjects, described by

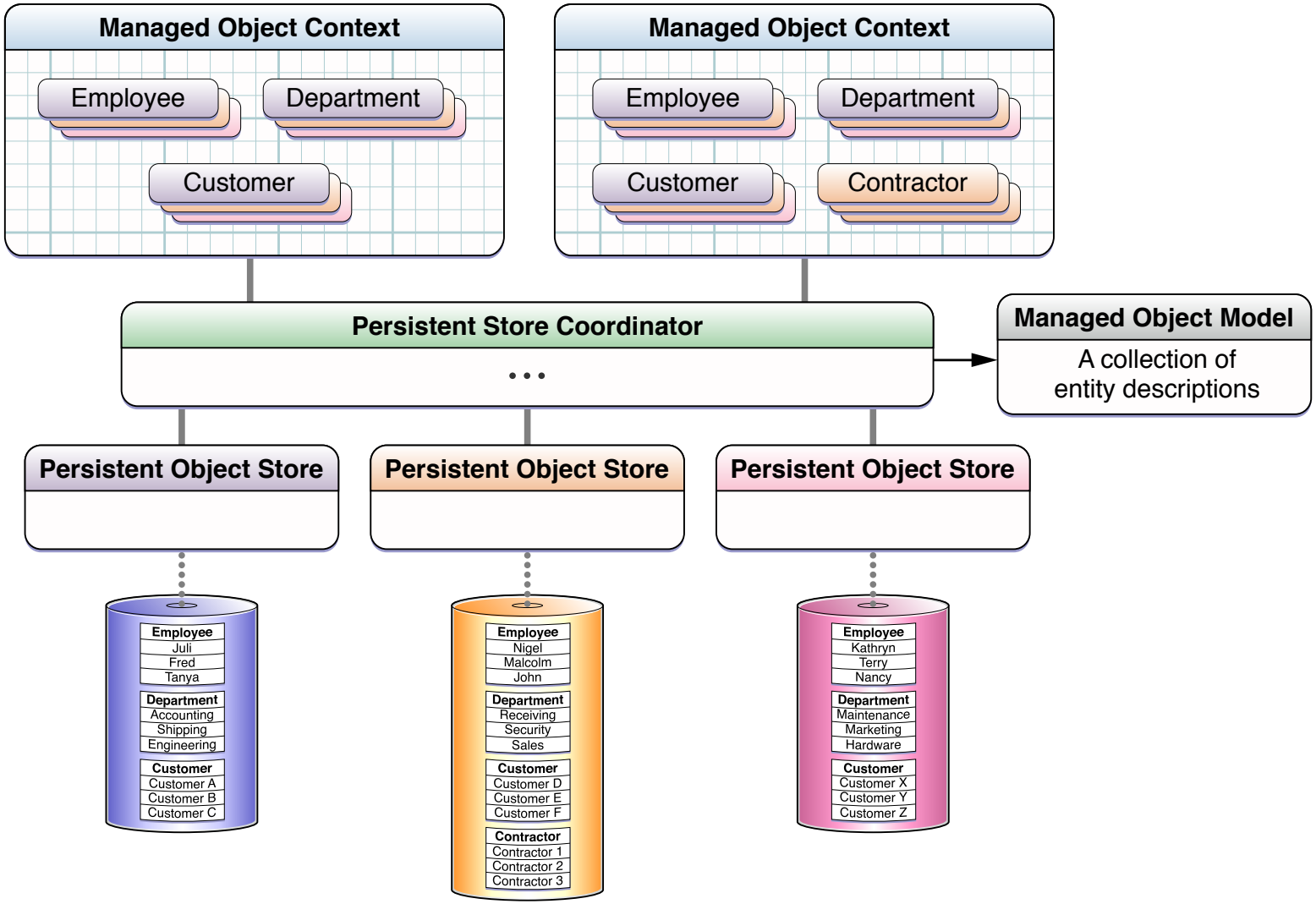
NSEntityDescriptions, live in a

NSManagedObjectContext, which acts as a scratchpad for a

NSPersistentStoreCoordinator, which persists an object graph to disk



Essential Core Data Stack



A Complex Core Data Setup

¶ Putting this together in code …

```
lazy var managedObjectModel: NSManagedObjectModel = {
  let modelURL = NSBundle.mainBundle().URLForResource("Model", withExtension: "momd")!
  return NSManagedObjectModel(contentsOfURL: modelURL)!
}()
```

```
lazy var persistentStoreCoordinator: NSPersistentStoreCoordinator? = {
  var coordinator: NSPersistentStoreCoordinator? =
        NSPersistentStoreCoordinator(managedObjectModel: self.managedObjectModel)
  let url = self.applicationDocumentsDirectory.URLByAppendingPathComponent(
        "Model.sqlite")
  var error: NSError? = nil
  coordinator!.addPersistentStoreWithType(NSSQLiteStoreType,
                  configuration: nil,
                  URL: url,
                  options: nil,
                  error: &error)
  return coordinator
}()
```

```
lazy var managedObjectContext: NSManagedObjectContext? = {
  let coordinator = self.persistentStoreCoordinator
  var managedObjectContext = NSManagedObjectContext()
  managedObjectContext.persistentStoreCoordinator = coordinator
  return managedObjectContext
}()
```

don't panic!

… all boilerplate code

typical setup: set up MOC in App Delegate & hand to root VC

```
func application(application: UIApplication, didFinishLaunchingWithOptions
    launchOptions: [NSObject: AnyObject]?) -> Bool {
  let navigationController = self.window!.rootViewController as UINavigationController
  let controller = navigationController.topViewController as MasterViewController
  controller.managedObjectContext = self.managedObjectContext
  return true
}
```

§ Working with Managed Objects

1. Creating them
2. Deleting them
3. Retrieving them
4. Managing large amounts of them

¶ Creating managed objects

need two things:

- entity description — *what* type of object
- managed object context — *where* to put it

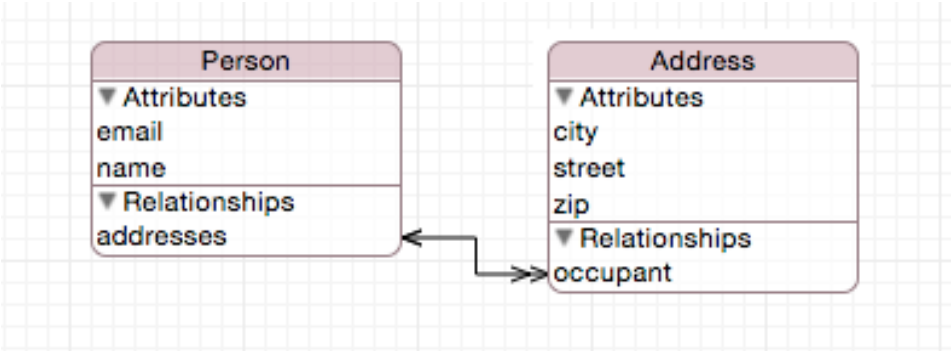
```swift
let entity = NSEntityDescription.entityForName("Person", inManagedObjectContext: moc)
let person = NSManagedObject(entity: entity!, insertIntoManagedObjectContext: moc)
```

```swift
let person = NSEntityDescription.insertNewObjectForEntityForName("Person",
                inManagedObjectContext: moc)
```

access managed object properties via Key-Value Coding (KVC):

```
let person: NSManagedObject = ...

let name = person.valueForKey("name") as String

person.setValue("John Doe", forKey: "name")
```

navigate/manipulate object graph:




```
let newAddress = NSEntityDescription.insertNewObjectForEntityForName("Address", ...)
newAddress.city = “Chicago"

person.mutableSetValueForKey("addresses").addObject(newAddress)

// OR

newAddress.setValue(person, forKey: "occupant")
```

the magic of KVC:

```
let personName   = address.valueForKeyPath("occupant.name") as String
let personCities = address.valueForKeyPath("occupant.addresses.city") as Set<String>
let numAddresses = address.valueForKeyPath("occupant.addresses.@count") as Int
```

modifying properties of a managed object *automatically informs the MOC* of the changes! (via Key-Value Observing)

… MOC may then notify other interested parties (e.g., VCs)

```
NSNotificationCenter.defaultCenter().addObserver(self,
  selector: "mgdObjChange:",
  name: NSManagedObjectContextObjectsDidChangeNotification,
  object: moc)
```

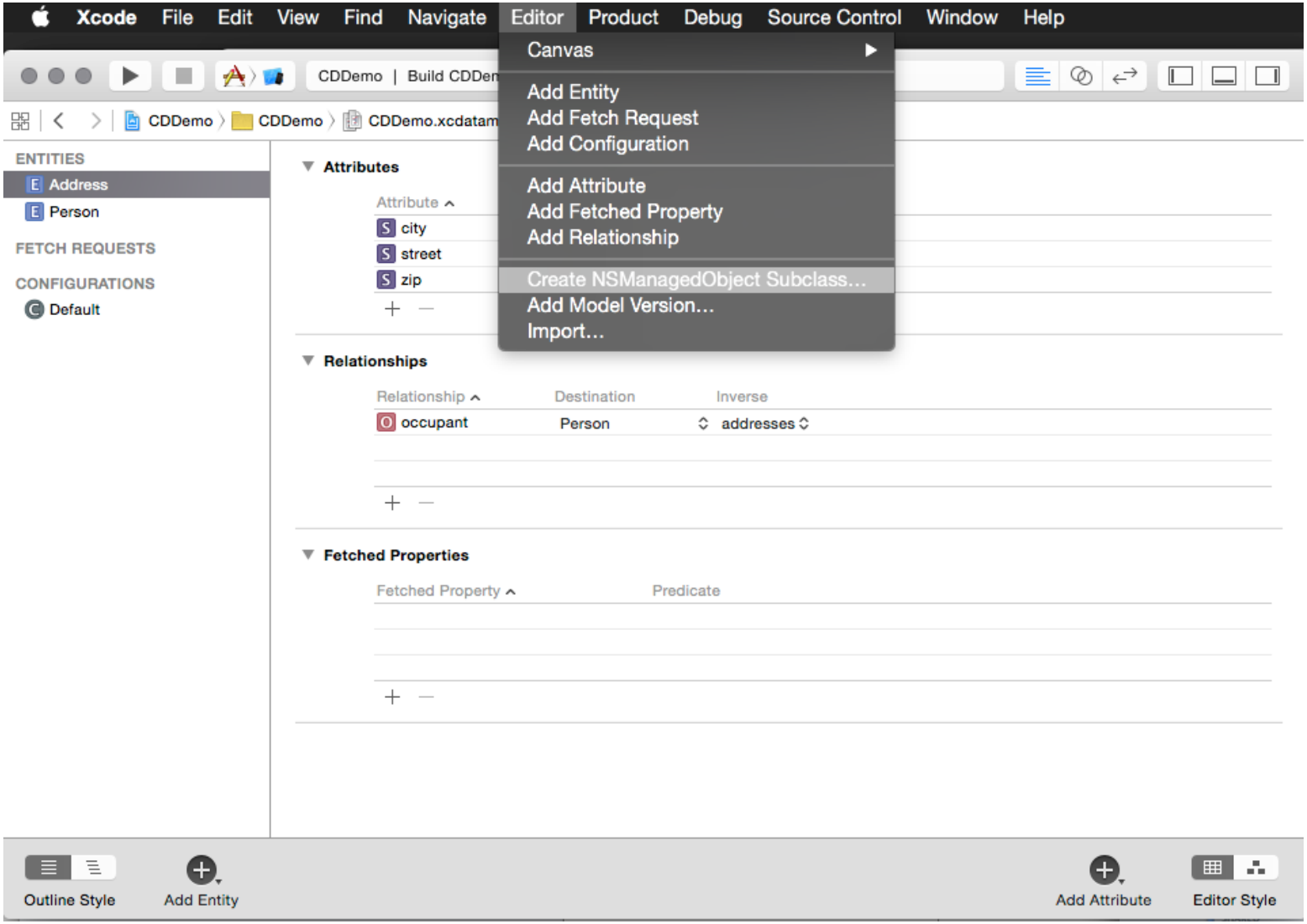
```
func mgdObjChange (notification: NSNotification) {
  for obj in notification.userInfo![NSUpdatedObjectsKey] as [NSManagedObject] {
    // do something with the changed obj (e.g., updated view)
  }
}
```

This syntax gets tiresome (and hard to read):

```
person.mutableSetValueForKey("addresses").addObject(newAddress)
```

Can prettify managed object access by subclassing NSManagedObject

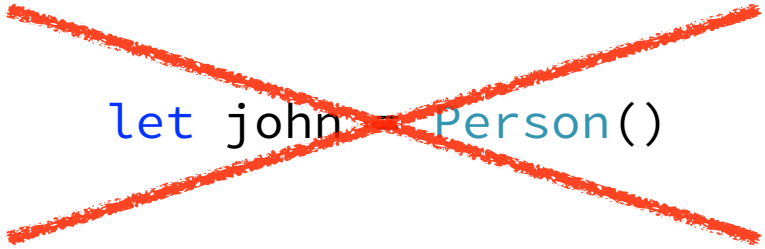
(not always necessary, but nice)


Xcode File Edit View Find Navigate Editor Product Debug Source Control Window Help
Canvas
Add Entity
Add Fetch Request
Add Configuration
Add Attribute
Add Fetched Property
Add Relationship
Create NSManagedObject Subclass…
Add Model Version…
Import…
CDDemo | Build CDDem
CDDemo CDDemo CDDemo.xcdatam
ENTITIES
Address
Person
FETCH REQUESTS
CONFIGURATIONS
Default
Attributes
Attribute
city
street
zip
Relationships
Relationship
Destination
Inverse
occupant
Person
addresses
Fetched Properties
Fetched Property
Predicate
Outline Style
Add Entity
Add Attribute
Editor Style

```
import Foundation
import CoreData

class Address: NSManagedObject {

    @NSManaged var street: String
    @NSManaged var city: String
    @NSManaged var zip: String
    @NSManaged var occupant: Person

}


class Person: NSManagedObject {

  @NSManaged var name: String
  @NSManaged var email: String
  @NSManaged var addresses: NSSet

  func addAddress (address: Address) {
    mutableSetValueForKey("addresses").addObject(address)
  }

}
```

note: all vars are “NSManaged” — i.e., computed values! (managed by CD)

```
let john  = NSEntityDescription.insertNewObjectForEntityForName("Person",
                inManagedObjectContext: moc) as Person
john.name = "John Doe"

let home  = NSEntityDescription.insertNewObjectForEntityForName("Address",
                inManagedObjectContext: moc) as Address
home.city = "Chicago"
home.occupant = john

let work  = NSEntityDescription.insertNewObjectForEntityForName("Address",
                inManagedObjectContext: moc) as Address
work.city = "Cupertino"
john.addAddress(work)

for addr in john.addresses as Set<Address> {
  println(addr.city)
}
```



Beware!!!

managed objects must be associated with a managed object context!

¶ Deleting managed objects

```
moc.deleteObject(person)
```

† may result in cascaded deletes!

¶ Retrieving managed objects

`NSFetchRequest`

is used to query and retrieve objects from the persistent store into an MOC

≈ database query / view

```swift
let fetchRequest = NSFetchRequest(entityName: "Person")

var error: NSError?
let fetchResults = moc.executeFetchRequest(fetchRequest, error: &error) as [Person]
```

```swift
let fetchRequest = NSFetchRequest(entityName: "Person")

fetchRequest.fetchBatchSize = 20 // no more than 20 returned at a time

var error: NSError?
let fetchResults = moc.executeFetchRequest(fetchRequest, error: &error) as [Person]
```

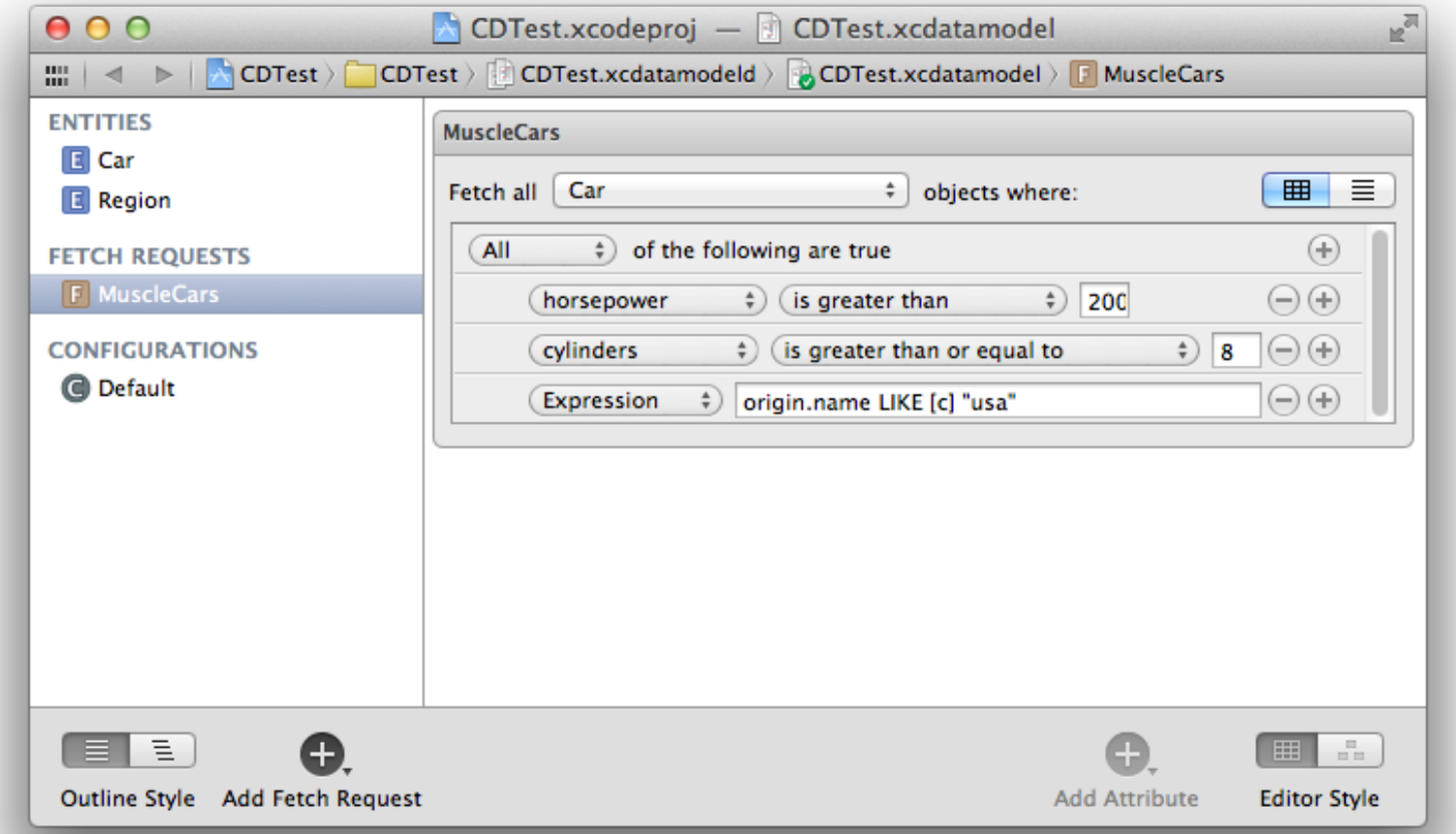
```swift
let fetchRequest = NSFetchRequest(entityName: "Person")

fetchRequest.fetchBatchSize = 20 // no more than 20 returned at a time

let sortDescriptor = NSSortDescriptor(key: "name", ascending: true)
fetchRequest.sortDescriptors = [sortDescriptor]

var error: NSError?
let fetchResults = moc.executeFetchRequest(fetchRequest, error: &error) as [Person]
```

```swift
let fetchRequest = NSFetchRequest(entityName: "Person")

fetchRequest.fetchBatchSize = 20 // no more than 20 returned at a time

let sortDescriptor = NSSortDescriptor(key: "name", ascending: true)
fetchRequest.sortDescriptors = [sortDescriptor]

let predicate = NSPredicate(format: "name contains %@", "ael")
fetchRequest.predicate = predicate // e.g., matches "Michael" and "Raphael"

var error: NSError?
let fetchResults = moc.executeFetchRequest(fetchRequest, error: &error) as [Person]
```

`NSPredicate`

is used to define logical expressions

e.g., for use in fetch request (or, previously, to filter an array)




```
let requestFromTemplate = managedObjectModel.fetchRequestTemplateForName("MuscleCars")!
let results = moc.executeFetchRequest(requestFromTemplate, error: &error)
```

- Managing large sets of fetched objects

e.g., in a tableview or navigation hierarchy

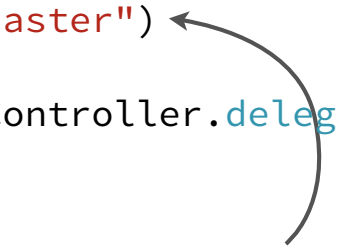
issues:

- mapping fetched results to cells and sections
- avoiding costly refetches (e.g., from DB)
- updating tableview when managed objects change (e.g., in another controller)

`NSFetchedResultsController`

wraps fetch request, maps objects to cells, and manages a persistent cache

```swift
let fetchRequest = NSFetchRequest(entityName: "Person")

let fetchedResultsController = NSFetchedResultsController(
  fetchRequest: fetchRequest,
  managedObjectContext: self.managedObjectContext!,
  sectionNameKeyPath: nil,
  cacheName: "Master")

fetchedResultsController.delegate = self // (view controller as delegate)
```



auto-persisted cache

```
 // (template code)

 override func numberOfSectionsInTableView(tableView: UITableView) -> Int {
   return self.fetchedResultsController.sections?.count ?? 0
 }

 override func tableView(tableView: UITableView,
       numberOfRowsInSection section: Int) -> Int {
   let sectionInfo = self.fetchedResultsController.sections![section]
   return sectionInfo.numberOfObjects
 }

 override func tableView(tableView: UITableView,
       cellForRowAtIndexPath indexPath: NSIndexPath) -> UITableViewCell {
   let cell = tableView.dequeueReusableCellWithIdentifier("Cell", forIndexPath: indexPath)
   self.configureCell(cell, atIndexPath: indexPath)
   return cell
 }

func configureCell(cell: UITableViewCell, atIndexPath indexPath: NSIndexPath) {
   let object = self.fetchedResultsController.objectAtIndexPath(indexPath) as NSManagedObject
   cell.textLabel!.text = object.valueForKey("name") as String
 }
```

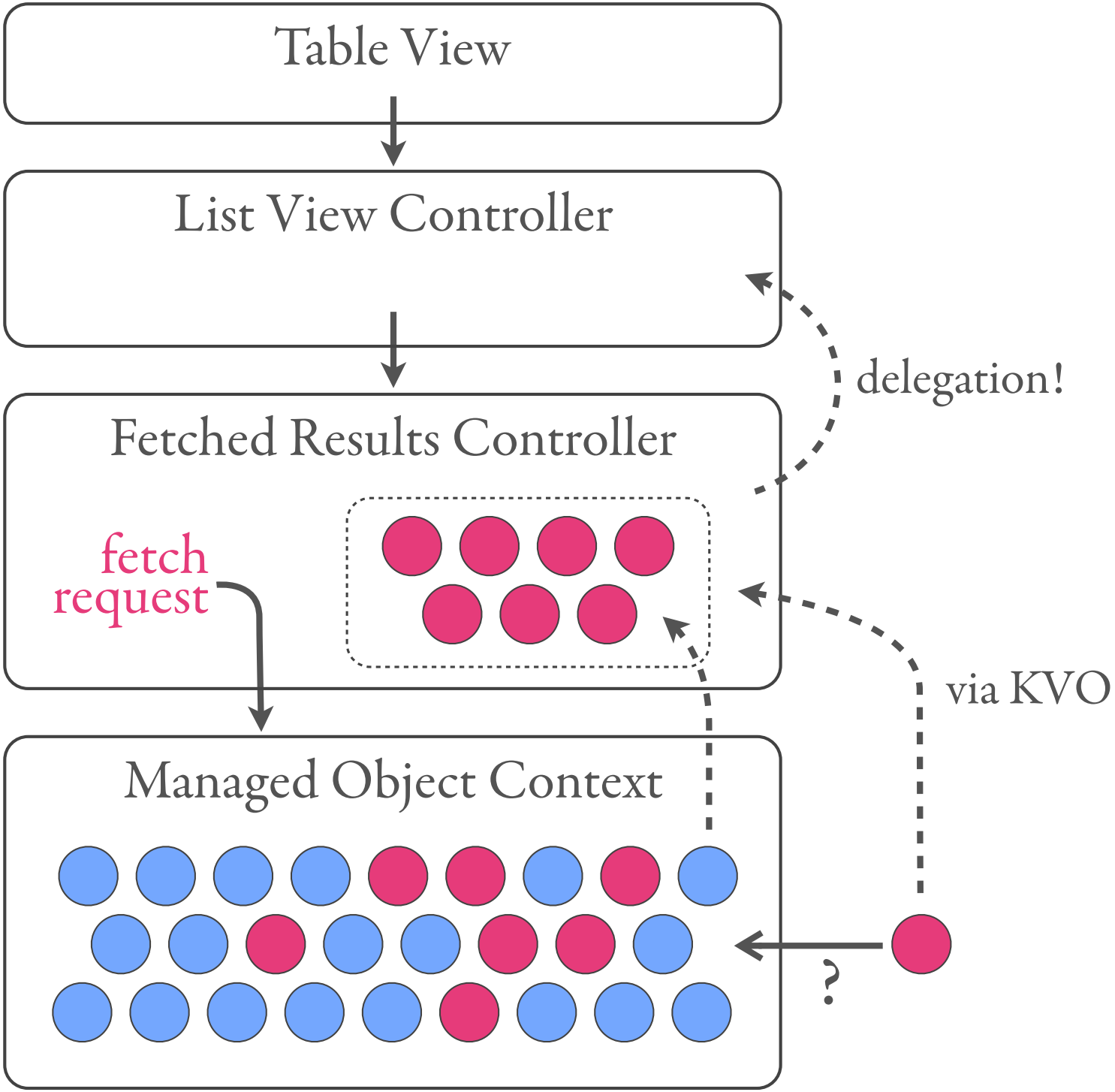


Table View
List View Controller
Fetched Results Controller
fetch request
delegation!
via KVO
Managed Object Context
?

`NSFetchedResultsController`

defines a delegate API for notifications of changes to associated fetch-results

```
@protocol NSFetchedResultsControllerDelegate
- (void)controllerWillChangeContent:(NSFetchedResultsController *)controller;

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

- (void)controllerDidChangeContent:(NSFetchedResultsController *)controller;
@end
```

(partial listing)

```swift
func controllerWillChangeContent(controller: NSFetchedResultsController) {
  self.tableView.beginUpdates()
}

func controller(controller: NSFetchedResultsController,
    didChangeObject anObject: AnyObject,
    atIndexPath indexPath: NSIndexPath?,
    forChangeType type: NSFetchedResultsChangeType,
    newIndexPath: NSIndexPath?) {
  switch type {
  case .Insert:
    tableView.insertRowsAtIndexPaths([newIndexPath!], withRowAnimation: .Fade)
  case .Delete:
    tableView.deleteRowsAtIndexPaths([indexPath!], withRowAnimation: .Fade)
  case .Update:
    self.configureCell(tableView.cellForRowAtIndexPath(indexPath!)!, atIndexPath: indexPath!)
  case .Move:
    tableView.deleteRowsAtIndexPaths([indexPath!], withRowAnimation: .Fade)
    tableView.insertRowsAtIndexPaths([newIndexPath!], withRowAnimation: .Fade)
  default:
    return
  }
}

func controllerDidChangeContent(controller: NSFetchedResultsController) {
  self.tableView.endUpdates()
}
```

§ Summary

NSManagedObjects, described by
NSEntityDescriptions, live in a
NSManagedObjectContext, which
acts as a scratchpad for a
NSPersistentStoreCoordinator, which
persists an object graph to disk

- core data adds overhead to persistence layer (e.g., SQLite)
- *but* also does caching & lazy-loading
- *and* adds uniquing, painless persistence, undo/redo + growing list of features

when to use?

- iOS 3.0+
- complex model objects
- dynamic & large object graph
- when able to use FRC with list views

when not to use?

- when portability (e.g., off iOS) is needed
- when small number of model objects
- when relational mapping is trivial (still debatable!)