Core Data Potpourri

Paul Goracke
paul@goracke.org
@pgor

What We Can Learn from an All-Night Marathon of Threes

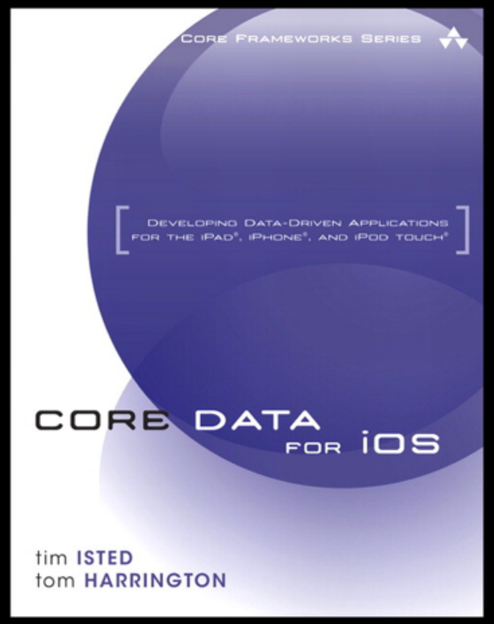
Paul Goracke
@pgor

Core Data Potpourri

Paul Goracke
paul@goracke.org
@pgor

“What I’m leaving out could fill a book.”

–Me

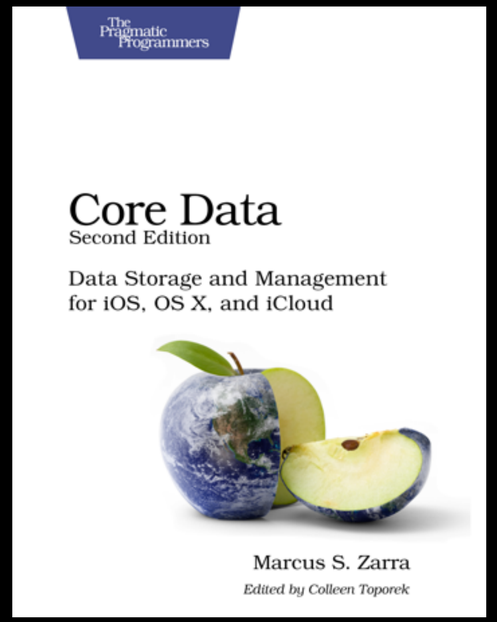


Core Data for iOS
Tim Isted and Tom Harrington

Addison-Wesley Professional
http://www.informit.com/
ISBN: 978-0-321-67042-7

Core Data (2nd Edition)
Marcus S. Zarra

Pragmatic Programmers
http://www.pragprog.com/
ISBN: 978-1-93778-508-6



My Biases

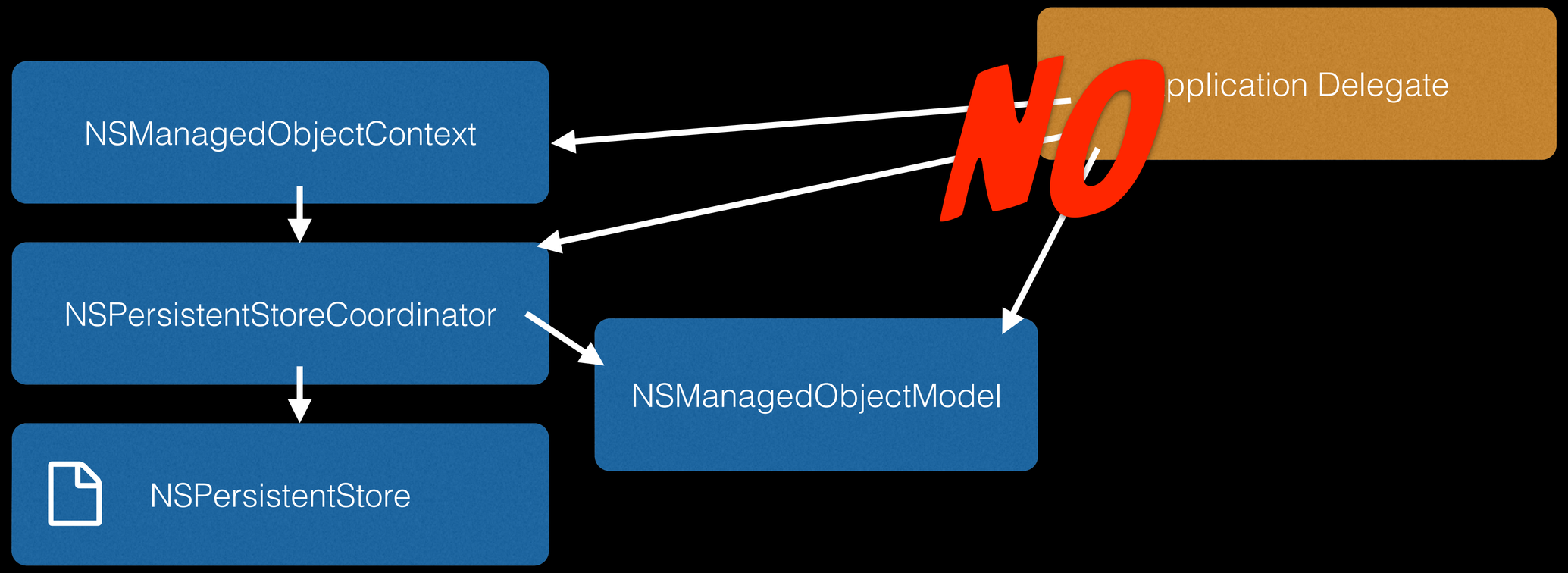
- I like static type-checking
- I dislike non-display strings in code
  - They lend themselves to typoes
    - Which lead to runtime errors, not compile time
      - Which means I have to exercise everything
    - They also slip by most refactoring
      - Which means I have to exercise everything. Again.
- Duplication of code leads to conflicts and errors

BEST PRACTICES

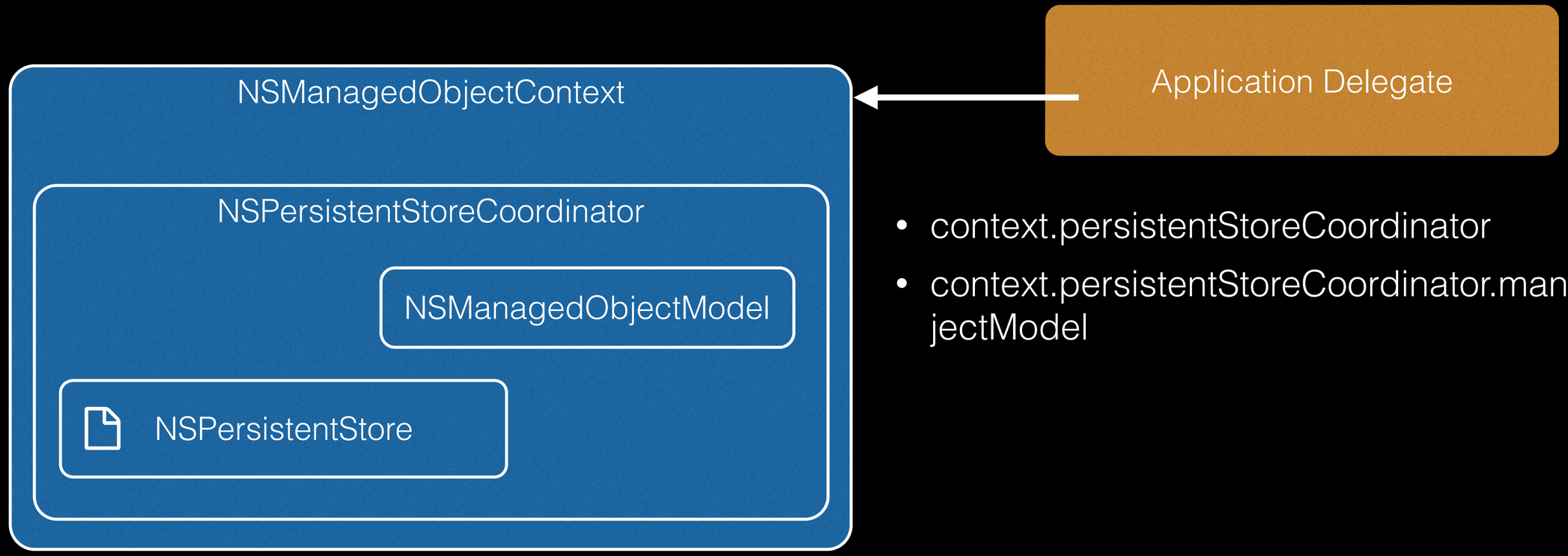
Prefix Your Entities

- You might be surprised how many models have Event or Location entities
  - If so, you might also be surprised how widely their definition of those entities can differ
- True, code samples don’t usually prefix
  - They aren’t real world; you are.
  - Be better than sample code.
- At some point you will be expected (or want) to merge models
  - Analytics libraries, for one

The Stack Is *One* Composed Object



The Stack Is *One* Composed Object



- context.persistentStoreCoordinator
- context.persistentStoreCoordinator.managedObjectModel

Subclass NSManagedObjectContext

```
@interface BPXLManagedObjectContext : NSManagedObjectContext

+ (instancetype) createAt:(NSURL*)storeURL;

@end
```

- Standalone, easy to create one or multiple
  - Unit testing
  - Command-line prebuild tool that doesn’t have to go through UIApplicationDelegate
- Separate subclasses per stack type enables static type-checking
- Add additional boilerplate code if desired

Subclass NSManagedObjectContext

```
+ (instancetype) createAt:(NSURL*)storeURL options:(NSDictionary*)options
{
   // Load Merged Models
   NSMutableArray * models = [NSMutableArray array];
   NSString* modelName = [self modelName];
   NSAssert( modelName != nil, @"%@ must implement +modelName", NSStringFromClass(self) );
   NSURL * modelURL = [[NSBundle mainBundle] URLForResource:modelName withExtension:@"momd"];
   NSAssert( modelURL != nil, @"Couldn't find model named '%@'", modelName );
   NSManagedObjectModel * model = [[NSManagedObjectModel alloc] initWithContentsOfURL:modelURL];
   NSAssert( model != nil, @"Couldn't load model at '%@'", modelURL );
   [models addObject:model];
   // Repeat for additional models…

   NSManagedObjectModel * model = [NSManagedObjectModel modelByMergingModels:models];
   NSAssert( model != nil, @"Could not load MOM" );

   // Instantiate PSC
   […]
   // Add persistent stores
   […]
   // Create context
   […]

   return context;
}
```

Subclass NSManagedObjectContext

```
+ (instancetype) createAt:(NSURL*)storeURL options:(NSDictionary*)options
{
   // Load Merged Models
   NSMutableArray * models = [NSMutableArray array];
   NSString* modelName = [self modelName];
   NSAssert( modelName != nil, @"%@ must implement +modelName", N
   NSURL * modelURL = [[NSBundle mainBundle] URLForResource:mode
   NSAssert( modelURL != nil, @"Couldn't find model named '%@'"
   NSManagedObjectModel * model = [[NSManagedObjectModel alloc]                    ];
   NSAssert( model != nil, @"Couldn't load model at '%@'", modelU
   [models addObject:model];
   // Repeat for additional models…

   NSManagedObjectModel * model = [NSManagedObjectModel modelByMergingModels:models];
   NSAssert( model != nil, @"Could not load MOM" );

   // Instantiate PSC
   […]
   // Add persistent stores
   […]
   // Create context
   […]

   return context;
}
```

+mergedModelFromBundles: may be more convenient, but provides less control

Context Creation Should Not Assume Location

- When you create multiple instances of your stack, would you expect them to all be in the same persistent store?
- Unit tests can create separate stacks without conflicts
- Simplifies writing an OS X tool to create a prebuilt store to include in the build process
- You may need multiple fileURL parameters for multiple stores

Subclass NSManagedObjectContext

```
+ (instancetype) createAt:(NSURL*)storeURL options:(NSDictionary*)options
{
   // Load Merged Models
   […]

   NSPersistentStoreCoordinator* coordinator =
      [[NSPersistentStoreCoordinator alloc] initWithManagedObjectModel:model];

   NSError * error = nil;
   NSString* storeType = NSSQLiteStoreType;

   if ( ! [coordinator addPersistentStoreWithType:storeType configuration:nil URL:storeURL
                                          options:options error:&error] ) {
      // TODO: Better error handling :)
      if ( [error code] == NSMigrationMissingSourceModelError ) {
         NSLog(@"Migration failed. Try deleting %@", storeURL); // For devs
      }
      else {
         NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
      }
      abort();
   }

   BPXLManagedObjectContext* context = [[self alloc] initWithPersistentStoreCoordinator:coordinator];

   return context;
}
```

Create an In-Memory Variant of Your Stack

- Incredibly useful for unit tests
  - Fast
  - No worries about file conflicts
  - Create in -setUp and have a fresh context every test

Create an In-Memory Variant of Your Stack

```
+ (instancetype) createAt:(NSURL*)storeURL options:(NSDictionary*)options
{
   // Load Merged Models
   […]

   NSPersistentStoreCoordinator* coordinator =
      [[NSPersistentStoreCoordinator alloc] initWithManagedObjectModel:model];

   NSError * error = nil;
   NSString* storeType = storeURL == nil ? NSInMemoryStoreType : NSSQLiteStoreType;

   if ( ! [coordinator addPersistentStoreWithType:storeType configuration:nil URL:storeURL
                                          options:options error:&error] ) {
      // TODO: Better error handling :)
      if ( [error code] == NSMigrationMissingSourceModelError ) {
         NSLog(@"Migration failed. Try deleting %@", storeURL); // For devs
      }
      else {
         NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
      }
      abort();
   }

   BPXLManagedObjectContext* context = [[self alloc] initWithPersistentStoreCoordinator:coordinator];

   return context;
}
```

Create an In-Memory Variant of Your Stack

```
+ (instancetype) createAt:(NSURL*)storeURL options:(NSDictionary*)options
{
   // Load Merged Models
   […]

   // Instantiate PSC
   […]
   // Add persistent stores
   […]
   // Create context
   […]

   return context;
}

+ (instancetype) createInMemory {
   return [self createAt:nil options:nil];
}
```

NSManagedObject Is the *Real* Top of the Stack

- Every MO belongs to one MOC
- It even has a reference to its MOC: .managedObjectContext
- Objects are invalid in any other MOC its own

NSManagedObject Is the *Real* Top of the Stack

```
+ (NSArray*) fetchSomethingForObject:(NSManagedObject*)object
                        withIdentifier:(NSString*)identifier
                 inManagedObjectContext:(NSManagedObjectContext*)context;

//++++++++++

- (instancetype) initViewControllerWithObject:(NSManagedObject*)object
                         inManagedObjectContext:(NSManagedObjectContext*);
```

- This is just begging for context != object.managedObjectContext
- What happens then?
  - Nothing that makes sense
  - Or a crash

NSManagedObject Is the *Real* Top of the Stack

```
+ (NSArray*) fetchSomethingForObject:(NSManagedObject*)object
                      withIdentifier:(NSString*)identifier;

//+++++++++++

- (instancetype) initViewControllerWithObject:(NSManagedObject*)object;
```

- Better
- Use the object’s managedObjectContext

Subclass NSManagedObject

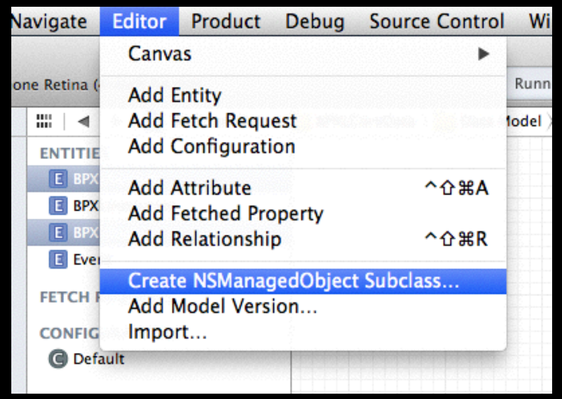
- Generic NSManagedObjects & KVC are neat but tedious:
  - “Stringly typed”
  - Easy typos
  - No autocomplete/discoverability
  - Difficult to refactor or rename
- No static type-checking
  - Don’t you want to know if you’re sending a Person object to a Department method?

Subclassing NSManagedObject

Subclassing NSManagedObject

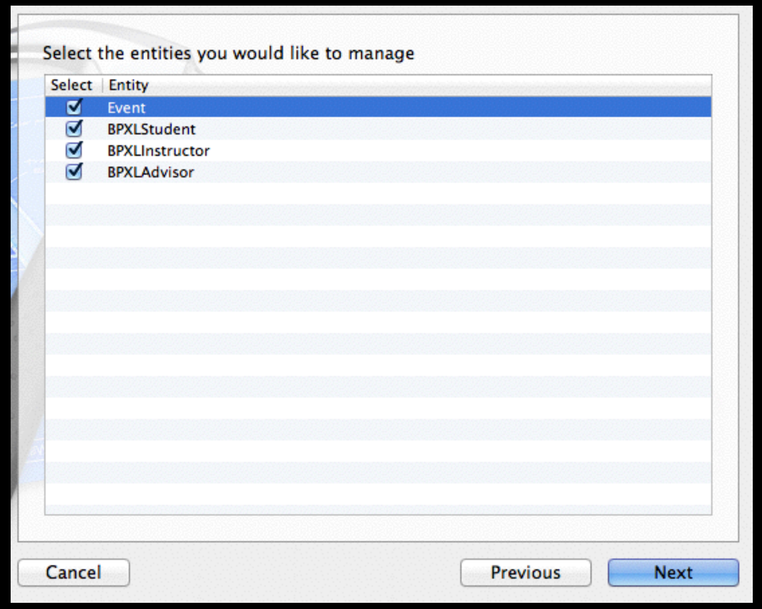
- Go to Model Editor (edit your xcdatamodel)

Subclassing NSManagedObject



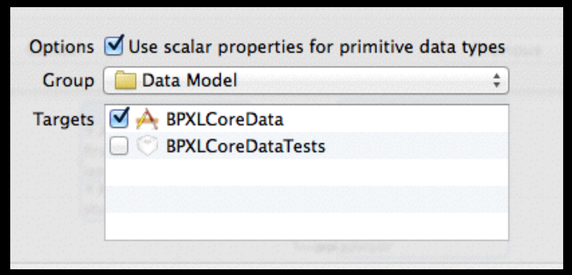
- Go to Model Editor (edit your xcdatamodel)
- Select “Editor > Create NSManagedObject Subclass…”

Subclassing NSManagedObject



- Go to Model Editor (edit your xcdatamodel)
- Select “Editor > Create NSManagedObject Subclass…”
- Select/check your model and entities to “manage” (i.e. create) and click “Next”

Subclassing NSManagedObject



- Go to Model Editor (edit your xcdatamodel)
- Select “Editor > Create NSManagedObject Subclass…”
- Select/check your model and entities to “manage” (i.e. create) and click “Next”
- Check “Use scalar properties for primitive data types”
  - Without this, number properties will be NSNumber, booleans will be NSValue
- Pay particular attention to the Group and file location

Voilà! A Header!

```
//  BPXLStudent.h

@class BPXLAdvisor;

@interface BPXLStudent : NSManagedObject
@property (nonatomic) NSTimeInterval birthDate;
@property (nonatomic, retain) NSString * firstName;
@property (nonatomic, retain) NSString * lastName;
@property (nonatomic, retain) BPXLAdvisor *advisor;
@end
```

- Properties are declared to match your entity attributes.
- Yes, they're "retain" not the new "strong". Change that if it bothers you.
- Some types should be changed:
  - NSDate comes out as NSTimeInterval. Change it (and add "retain").
  - NSInteger instead of int types, CGFloat instead of floats, etc.

Voilà! An Implementation!

```
//  BPXLStudent.m
[…]

@implementation BPXLStudent

@dynamic birthDate;
@dynamic firstName;
@dynamic lastName;
@dynamic advisor;

@end
```

- Properties are declared @dynamic, not @synthesized
  - The getters and setters were there all along at runtime, this just exposes them for autocompletion and compilation checking
- According to Apple, using these accessors is faster than plain KVC

Goodbye, Key Strings

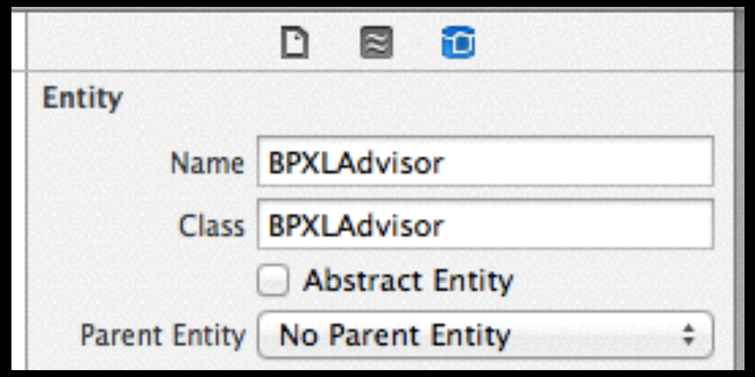
```
[student setValue:@“Howard" forKey:@"firstName"];

// becomes

student.firstName = @“Howard”;
```

Extras for To-Many Relationships

```
@interface BPXLAdvisor : NSManagedObject
[…]
@property (nonatomic, retain) NSSet *students;
@end

@interface BPXLAdvisor (CoreDataGeneratedAccessors)
- (void)addStudentsObject:(NSManagedObject *)value;
- (void)removeStudentsObject:(NSManagedObject *)value;
- (void)addStudents:(NSSet *)values;
- (void)removeStudents:(NSSet *)values;
@end
```

- Relationship is an NSSet
- The CoreDataGeneratedAccessors category declare KVC-standard mutable unordered accessors -add<Key>Object:, -remove<Key>Object:, -add<Key>:, and -remove<Key>:.
- Again, this is just exposing methods to the compiler that will exist at runtime

The Entity's Class Is Now Defined



- The MOM now knows to create instances of the specified class instead of generic NSManagedObject when creating this entity type.

Why Not mogenerator?

- The human/machine separation is a smart mechanism to avoid losing your customizations. But:
  - It adds an additional level of class hierarchy on the assumption that the model will change frequently enough to be inconvenient
  - It presents object and scalar versions of properties, effectively doubling the interface
  - I feel model changes *should* be considered and deliberate, not automatic
  - It does provide some handy convenience methods, but we can easily replicate those in a common base class
  - http://rentzsch.github.io/mogenerator/

Why Not a Category on NSMO?

- Again, it adds a secondary level
  - Does not provide runtime type-checking
  - Client code needs to know to import “MyNSMO+Custom.h”, not just “MyNSMO.h”
  - As long as we can control the class that is generated, why make the class more complicated?
  - Again, it’s geared around saving time changing the model, not ease and clarity of code

Create a Common NSManagedObject Base Class

```
@interface BPXLManagedObject : NSManagedObject
@end

@implementation BPXLManagedObject

+ (NSString*) entityName {
  // subclasses can override, but provide a reasonable default
  return NSStringFromClass(self);
}

@end
```

- Goodbye entityName string all over the place

Create a Common NSManagedObject Base Class

```
+ (instancetype) newInstanceInManagedObjectContext:(NSManagedObjectContext*)context {
   id object =
     [NSEntityDescription insertNewObjectForEntityForName:[self entityName]
                                   inManagedObjectContext:context];
   return object;
}
```

- Simply a wrapper of the standard creation method
- But leverages -entityName so client code doesn't even need to know it, just the class name
- Now we're starting to get back to the *object* persistence promise of Core Data!

Simpler Code

```
NSManagedObject* managedObject =
   [NSEntityDescription insertNewObjectForEntityForName:@"MyEntity"
                                   inManagedObjectContext:moc];
```

...becomes...

```
MyEntity* myEntity = [MyEntity newInstanceInManagedObjectContext:moc];
```

Move Predicates and Requests Into the Model

```
// NSMO Base Class

+ (NSFetchRequest*) fetchRequest {
   return [NSFetchRequest fetchRequestWithEntityName:[self entityName]];
}

+ (NSArray*) allInstancesWithPredicate:(NSPredicate*)predicate
               inManagedObjectContext:(NSManagedObjectContext*)context
{
   NSFetchRequest* request = [self fetchRequest];
   request.predicate = predicate;
   NSError* error = nil;
   NSArray* results = [context executeFetchRequest:request error:&error];
   if ( results == nil ) {
      NSLog(@"ERROR loading %@: %@", predicate, error);
   }

   return results;
}

+ (NSArray*) allInstancesInManagedObjectContext:(NSManagedObjectContext*)context {
   NSArray* results = [self allInstancesWithPredicate:nil inManagedObjectContext:context];
   return results;
}
```

Move Predicates and Requests Into the Model

```
// BPXLCourse.m (Custom NSMO Class)

+ (NSArray*) coursesAvailableForMajor:(BPXLStudiesMajor*)major
                              quarter:(BPXLAcademicQuarter*)quarter
                inManagedObjectContext:(NSManagedObjectContext*)context;

+ (BPXLCourse*) courseWithIdentifer:(BPXLCourseIdentifier*)identifier;

- (NSArray*) quartersOffered;
```

- Build predicates from the passed parameters
- Invoke base class query methods

Move Predicates and Requests Into the Model

- Client code can operate in its problem domain
- Centralized queries are easier to test
  - And optimize
- Some customization via parameters may be needed
  - e.g., sort descriptors

Defining Property Constants with Accessorizer

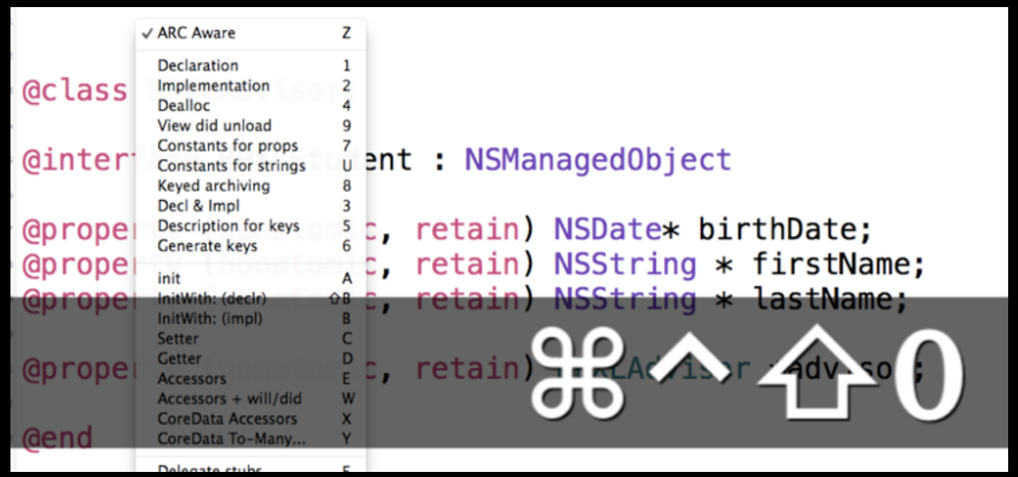
Defining Property Constants with Accessorizer

- Select entire @interface declaration block and invoke Accessorizer with ⌘⌥⇧0

```
@interface BPXLStudent : NSManagedObject

@property (nonatomic, retain) NSDate* birthDate;
@property (nonatomic, retain) NSString * firstName;
@property (nonatomic, retain) NSString * lastName;

@property (nonatomic, retain) BPXLAdviser *adviser;

@end
```

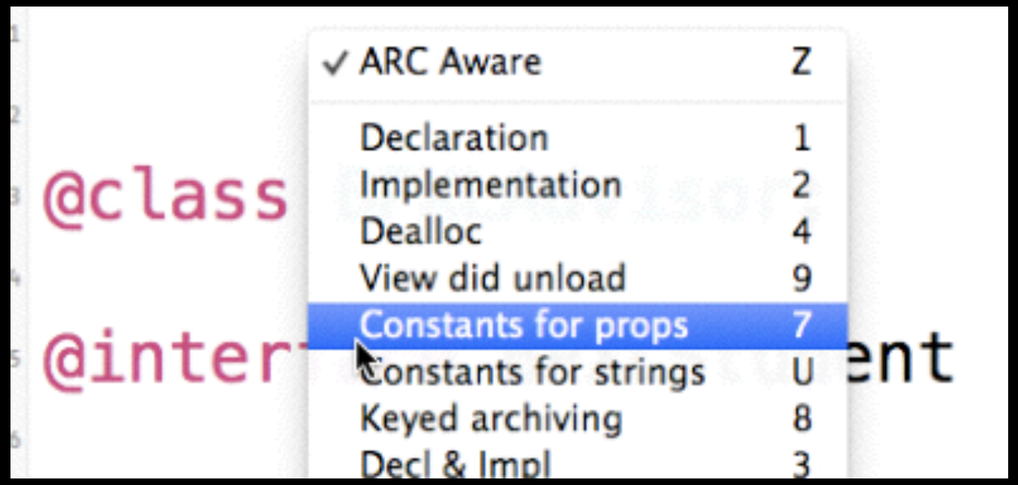
⌘⌥⇧0

Defining Property Constants with Accessorizer



- Select entire @interface declaration block and invoke Accessorizer with ⌘⌥⇧0
- Move insertion point to desired location and press ⌘⌃⇧0

Defining Property Constants with Accessorizer



- Select entire @interface declaration block and invoke Accessorizer with ⌘⌥⇧0
- Move insertion point to desired location and press ⌘^⇧0
- Select “Constants for props” from the contextual menu

Defining Property Constants with Accessorizer

```
NSString* const BPXLStudentKeyBirthDate = @"birthDate";
NSString* const BPXLStudentKeyFirstName = @"firstName";
NSString* const BPXLStudentKeyLastName = @"lastName";
NSString* const BPXLStudentKeyAdvisor = @"advisor";


extern NSString* const BPXLStudentKeyBirthDate;
extern NSString* const BPXLStudentKeyFirstName;
extern NSString* const BPXLStudentKeyLastName;
extern NSString* const BPXLStudentKeyAdvisor;

@class BPXLAdvisor;
```

- Select entire @interface declaration block and invoke Accessorizer with ⌘⌥⇧0
- Move insertion point to desired location and press ⌘^⇧0
- Select “Constants for props” from the contextual menu

Defining Property Constants with Accessorizer

```
NSString* const BPXLStudentKeyBirthDate = @"birthDate";
NSString* const BPXLStudentKeyFirstName = @"firstName";
NSString* const BPXLStudentKeyLastName = @"lastName";
NSString* const BPXLStudentKeyAdvisor = @"advisor";

@implementation BPXLStudent
```

- Select entire @interface declaration block and invoke Accessorizer with ⌘⌥⇧0
- Move insertion point to desired location and press ⌘^⇧0
- Select “Constants for props” from the contextual menu
- Cut and paste the non-extern statements into .m file
- http://www.kevincallahan.org/software/accessorizer.html

Use String Constants In Your Predicate Formats

```
request.predicate = [NSPredicate predicateWithFormat:@“%K = %@“,
                                    BPXLStudentKeyFirstName, firstName];

(lldb) po [request predicate]
firstName == "Howard"
```

- If BPXLStudentKeyFirstName changes, the predicate is automatically updated
- If the property goes away completely, I am alerted by a compiler error
- Property names are often overloaded throughout a project. A search will no longer merge together this usage of @“firstName” with one parsing incoming JSON data

Your Data Model Should Be Opinionated

- Its design should be driven by app usage design
  - What lists are needed, and how many items will there be?
  - What items will be detailed from this list?
- Server feeds suck
  - (except the ones you implemented, of course)
- They provide the *data*; you are providing a way of *viewing* that data
- You will never create a highly responsive model if you tie yourself in lockstep to the server feed data design
- *Transform* the data
  - Break input down to searchable parts as needed
  - Don’t keep data you don’t need

Parsing Is Not the Model's Responsibility

```
// NSManagedObject subclass

- (void) parseWithDictionary:(NSDictionary*)json {

}
```

- What will you name the method for parsing a slightly different data feed?
- And the next one?
- Changes to the data model ripple throughout the app
  - A well-tested model should not have to risk change because a data feed changed or was added
- Make your parsers vend well-defined MOs

Parsing Is Not the Model's Responsibility

```
// NSManagedObject subclass

- (void) parsWithDictionary:(NSDictionary*)json {

}
```

NO

- What will you name the method for parsing a slightly different data feed?
- And the next one?
- Changes to the data model ripple throughout the app
  - A well-tested model should not have to risk change because a data feed changed or was added
- Make your parsers vend well-defined MOs

Formatting Is Not the Model's Responsibility

```
- (NSString*) registrationDateISO8601 {
}

- (NSString*) registrationDateMMDDYYY {
}

- (NSString*) registrationDateVerbose {
}
```

- An MVC data model is intended to allow many different views of the same data
- That doesn’t mean it needs to know about them
- Formatting at the view controller allows reuse of formatter (e.g. table views) and existence of formatters only when actively in use
- Same for parsing strings from data feeds

Saving Is *Not* a Model Object's Responsibility

```
+ (BPXLStudent) findOrCreateStudentWithIdentifier:(NSString*)identifier
   inManagedObjectContext:(NSManagedObjectContext*)context
{
   BPXLStudent* student = // search
   if ( student == nil ) {
      student = [BPXLStudent newInstanceInManagedObjectContext:context
      student.identifier = identifier;

      [context save:nil];
   }

   return student;
}
```

- (Yes, I usually encounter it with "save:nil")
- Any required constraints are almost definitely not set up yet
  - Constraints such as "student must have a related cafeteria account/record"
- You aren't even at a high enough level to handle validation errors

Save a Context at the Highest Level Possible

- The only code that can determine whether it’s ready to save is the code that started the edit
- Importer/Parser: At the end of the parsed document
  - Possibly at intervals during the parse
  - Even at intervals, it’s the only place that knows to expect valid data

Plan for Scale

- Prevent scale
  - Read-only stores
  - Constant pruning
- Accommodate scale
  - Tune performance to handle large scale, even outliers
  - Beware of migrations

“Was a great app, but latest update starts up then quits. EVERY TIME. Can’t access all the data I’d entered. Deleted, 1 Star.”

–App Store customers of developers who didn’t consider migrations

What Happened?

- App instantiates MOC in -application:didFinishLaunchingWithOptions:
- Updated app requires migration, which needs to happen before accessing the MOC
- Enough data for the migration to take too long, triggering the Watchdog
- Next launch, tries to migrate again…

Add a Migration Checker

```
// Your NSManagedObjectContext base class

+ (BOOL) storeNeedsMigrationAtURL:(NSURL*)sourceStoreURL {
   BOOL compatible = NO;
   NSError *error = nil;

   NSDictionary *sourceMetadata =
   [NSPersistentStoreCoordinator metadataForPersistentStoreOfType:nil
                                                            URL:sourceStoreURL
                                                          error:&error];
   if ( sourceMetadata != nil ) {
      NSManagedObjectModel *destinationModel = [self loadManagedObjectModel];
      compatible = [destinationModel isConfiguration:nil
                          compatibleWithStoreMetadata:sourceMetadata];
   }
   else {
      // deal with error?
   }

   return ! compatible;
}
```

Add a Migration Checker

```
// Your App Delegate

- (BOOL) application:(UIApplication *)application
 didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
   […]
   NSURL *sourceStoreURL = // path to store;
   if ( [BPXLManagedObjectContext storeNeedsMigrationAtURL:sourceStoreURL] ) {
      // Change the interface to a “Please wait, migrating…” UI
      // Dispatch migration to the background and restore interface when done
   }
   else {
      controller.managedObjectContext = [BPXLManagedObjectContext createAt:sourceStoreURL];
   }

   […]
}
```

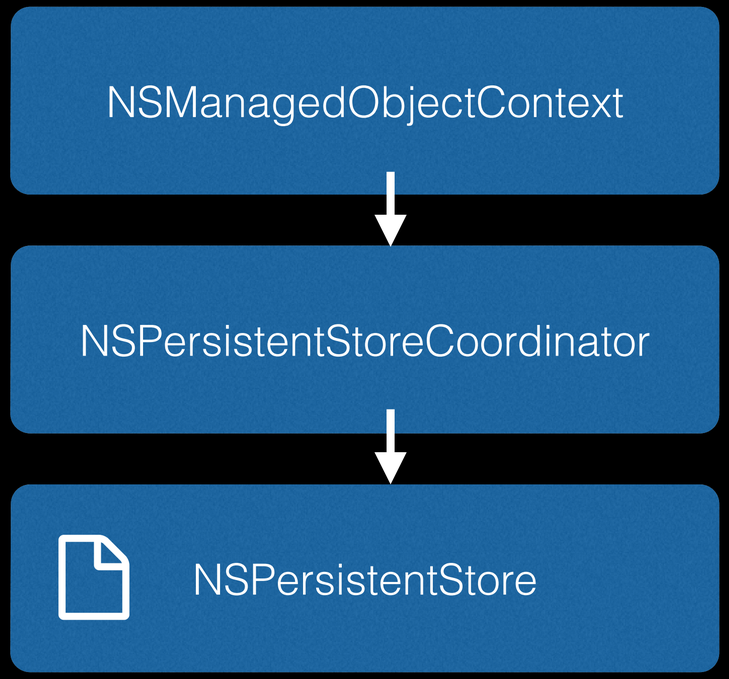
- Migration is left as an exercise to the reader (of Isted’s or Zarra’s books)

Review

- Prefix Your Entities
- The Stack Is *One* Composed Object
- Subclass NSManagedObjectContext
- Context Creation Should Not Assume Location
- Create an In-Memory Variant of Your Stack
- NSManagedObject Is the *Real* Top of the Stack
- Subclass NSManagedObject
- Create a Common NSManagedObject Base Class
- Move Predicates and Requests Into the Model
- Your Data Model Should Be Opinionated
- Saving Is *Not* a Model Object's Responsibility
- Plan for Scale

Multi-Threading

Multi-Threading Without Parent-Child Contexts



- Contexts may not be thread-safe, but they know how to share a PSC in a thread-safe manner
  - (NSPersistentStoreCoordinator is not thread-safe on its own, either)
- UI work must be done on main thread, so at needs to use a context created on main
- So create a secondary “editing” context (once on the target background thread) that points uses the shared PSC
  - Edit
  - Save
  - Update original context

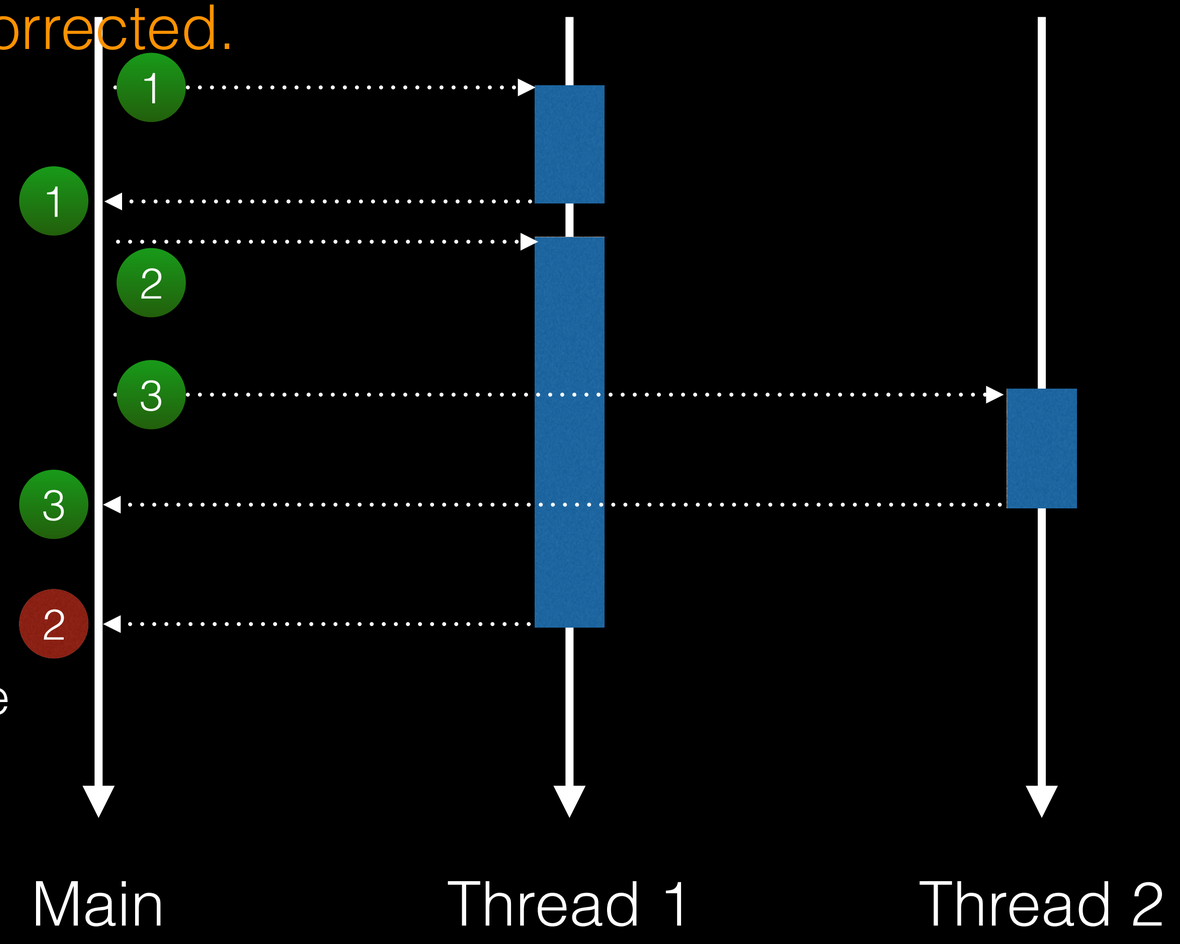
Everybody Loves GCD (Part 1)

```
NSPersistentStoreCoordinator* psc =
   self.managedObjectContext.persistentStoreCoordinator;
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
   NSManagedObjectContext* context = [[NSManagedObjectContext alloc] init];
   context.persistentStoreCoordinator = psc;
   // do stuff
   NSError* error = nil;
   BOOL saved = [context save:&error];
   if ( ! saved ) {
      NSLog(@"ERROR saving in background: %@", error);
   }
});
```

- The Good
  - I'm not using threadDictionary
  - I am treating the context as a transaction (saving at the highest level)
- The Bad
  - Although the changes are saved via the PSC, the original MOC doesn't know of the changes
  - Dispatch queues are concurrent

How Can Concurrent Be Bad?

NOTE: This numbering is wrong.
Needs to be corrected.



Optimistic Locking Failure

Treat Contexts As Transactions

```
BEGIN TRANSACTION

Do lots of stuff: Creating, deleting, modifying

END TRANSACTION
```

Overview

- One master context on main queue, used by UI
  - mergePolicy = NSMergeByPropertyStoreTrumpMergePolicy
- An NSOperationQueue, .maxConcurrentOperations = 1
- NSOperation objects
  - In -init, grab main context
  - In -main, create a context based off main context, with shared PSC
    - Observe NSManagedObjectContextDidSaveNotification and dispatch to main context's -mergeChangesFromContextDidSaveNotification:
  - mergePolicy = NSMergeByPropertyObjectTrumpMergePolicy