Stanford CS193p

Developing Applications for iPhone 4, iPod Touch, & iPad

Fall 2010

Review

- Objective-C
  Classes, Methods, Properties, Protocols, Delegation, Memory Management
- Foundation
  `NSArray`, `NSDictionary`, `NSString` (and mutable versions thereof)
- MVC and `UIViewController`
  Separation of Model from View using Controller
  `view` property in `UIViewController` (set in `loadView` or from `.xib`)
  `viewDidLoad`, `viewDidUnload` (outlet releasing), orientation changes
  `viewWillAppear:`/`Disappear:`, `title`, `initWithNibName:bundle:`/`awakeFromNib`
- Interface Builder
  Creating a View using drag and drop and setting properties (including autosizing) via the Inspector
  Custom view by dragging a generic `UIView`, then changing the class via the Inspector
  `UIButton`, `UILabel`, `UISlider`

Review

- Custom Views

  drawRect:, UIGestureRecognizer, initWithFrame:/awakeFromNib, Core Graphics (CGContext...)

- Application Lifecycle

  application:didFinishLaunchingWithOptions:, MainWindow.xib

  Platform-conditional code for iPad versus iPhone/iPod-Touch

  Running application in 3.2 or 4.0 simulators

- UINavigationController

  Pushing onto the stack, navigationItem method in UIViewController

- Controllers of Controllers

  UITabBarController, tabBarItem method in UIViewController

  UISplitViewController, delegate methods to handle the bar button in portrait mode

  UIPopoverController (not a UIViewController)

Review

- `UIScrollView`

  `contentSize` property, understanding that its `bounds` is its visible area (like any view)

- `UIImageView`, `UIWebView`

  `UIView`s for displaying images or web content

- `UITableView`

  Styles (plain and grouped)

  `UITableViewDataSource` (number of sections, rows, and loading up a cell to display)

  `UITableViewCell` (how it is reused, properties that can be set on it)

  Displaying content in section headers/footers.

  Editing (i.e. deleting or inserting).

  `UITableViewDelegate` (other customization for the table via a delegate)

Today: Persistence

- Property Lists
- Archiving Objects
- Storing things in the Filesystem
- SQLite
- Core Data

Property Lists

- Any combination of the following classes:

  `NSArray`, `NSDictionary`, `NSString`, `NSData`, `NSDate`, `NSNumber`.

- Only good for small amounts of data

  You would never want your application's actual "data" stored here.
  Good for "preferences" and "settings."

- Can be stored permanently

  `NSUserDefaults`

  Also three formats for storing in files or reading from internet via a URL:

  XML

  Binary

  "Old-style" ASCII (deprecated) … good for demos only :).

NSPropertyListSerialization

- Creating an NSData from a plist w/NSPropertyListSerialization

```
+ (NSData *)dataFromPropertyList:(id)plist
                          format:(NSPropertyListFormat)format    // XML or Binary
                         options:(NSPropertyListWriteOptions)options // unused, set to 0
                           error:(NSError **)error;
```

- Creating a plist from an NSData blob

```
+ (id)propertyListWithData:(NSData *)plist
                   options:(NSPropertyListReadOptions)options // see below
                    format:(NSPropertyListFormat *)format    // returns XML or Binary
                     error:(NSError **)error;
```

- NSPropertyListReadOptions

```
NSPropertyListImmutable
NSPropertyListMutableContainers           // an array of arrays would BOTH be mutable
NSPropertyListMutableContainersAndLeaves  // an array of strings would have mutable strings
```

Property Lists

- To write a property list to a file …

  Use NSPropertyListSerialization to get an NSData, then use this NSData method …

  ```
  + (BOOL)writeToURL:(NSURL *)fileURL atomically:(BOOL)atomically;
  ```

  Returns whether it succeeded.

  Specifying atomically means it will write to a temp file, then move it into place.

  Only file URLs (i.e. "file://") are currently supported.

- To read a property list NSData from a URL

  Get the NSData from a URL …

  ```
  + initWithContentsOfURL:(NSURL *)aURL;
  ```

  Then use NSPropertyListSerialization to turn the NSData back into a property list.

  NSData can read from a non-file URL (e.g. a web server) …

  But it's more likely you'd be reading property lists in an industry-standard format like JSON.

  Then converting that to property lists (e.g. using the library included in your homework).

Property Lists

- What does the XML property list format look like?

  This is a dictionary with a single key "y = 0.5x" whose value is an array (0.5, "*", "@x", "=").

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
                "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
   <key>y = 0.5x</key>
   <array>
      <real>0.5</real>
      <string>*</string>
      <string>@x</string>
      <string>=</string>
   </array>
</dict>
</plist>
```

Archiving

- There is a mechanism for making ANY object graph persistent
  Not just graphs with `NSArray`, `NSDictionary`, etc. in them.
- For example, the view hierarchies you build in Interface Builder
  Those are obviously graphs of very complicated objects.
- Requires all objects in the graph to implement `NSCoding` protocol
  - `- (void)encodeWithCoder:(NSCoder *)coder;`
  - `- initWithCoder:(NSCoder *)coder;`

Archiving

- Object graph is saved by sending all objects `encodeWithCoder:`

```
- (void)encodeWithCoder:(NSCoder *)coder {
    [super encodeWithCoder:coder];
    [coder encodeFloat:scale forKey:@“scale”];
    [coder encodeCGPoint:origin forKey:@“origin”];
    [coder encodeObject:expression forKey:@“expression”];
}
```

Absolutely, positively must call `super`’s version or your superclass’s data won’t get written out

- Object graph is read back in with `alloc`/`initWithCoder:`

```
- initWithCoder:(NSCoder *)coder {
    self = [super initWithCoder:coder];
    scale = [coder decodeFloatForKey:@“scale”];
    expression = [[coder decodeObjectForKey:@“expression”] retain]; // note retain!
    origin = [coder decodeCGPointForKey:@“origin”];  // note that order does not matter
}
```

Archiving

- NSKeyed[Un]Archiver classes used to store/retrieve graph

  Storage and retrieval is done to NSData objects.

  NSKeyedArchiver stores an object graph to an NSData …

```
+ (NSData *)archivedDataWithRootObject:(id <NSCoder>)rootObject;
```

  NSKeyedUnarchiver retrieves an object graph from an NSData …

```
+ (id <NSCoder>)unarchiveObjectWithData:(NSData *)data;
```

- What do you think this code does?

```
id <NSCoder> object = ...;
NSData *data = [NSKeyedArchiver archivedDataWithRootObject:object];
id <NSCoder> dup = [NSKeyedArchiver unarchiveObjectWithData:data];
```

  It makes a "deep copy" of object.

  But beware, you may get more (or less) than you bargained for.

  Object graphs like "view hierarchies" can be very complicated.

  For example, does a view's superview get archived?

File System

- Your application sees iOS file system like a normal Unix filesystem
  It starts at /.
  There are file protections, of course, like normal Unix, so you can't see everything.
- You can only WRITE inside your "sandbox"
- Why?
  Security (so no one else can damage your application)
  Privacy (so no other applications can view your application's data)
  Cleanup (when you delete an application, everything its ever written goes with it)
- So what's in this "sandbox"
  Application bundle directory (binary, `.xibs`, `.jpgs`, etc.). This subdirectory is NOT writeable.
  Documents directory. This is where you store permanent data created by the user.
  Caches directory. Store temporary files here (this is not backed up by iTunes).
  Other directories (check out `NSSearchPathDirectory` in the documentation).

File System

- What if you want to write to a file you ship with your app?
  Copy it out of your application bundle into the documents (or other) directory so its writeable.
- How do you get the paths to these special sandbox directories?

```
NSArray *NSSearchPathForDirectoriesInDomains(
    NSSearchPathDirectory directory,     // see below
    NSSearchPathDomainMask domainMask,   // NSUserDomainMask
    BOOL expandTilde                     // YES
);
```

- Notice that it returns an `NSArray` of paths (not a single path)
  Since the file system is limited in scope, there is usually only one path in the array in iOS.
  No user home directory, no shared system directories (for the most part), etc.
  Thus you will almost always just use `lastObject` (for simplicity).
- Examples of `NSSearchPathDirectory` values
  `NSDocumentsDirectory`, `NSCachesDirectory`, `NSAutosavedInformationDirectory`, etc.

File System

- NSFileManager

  Provides utility operations (reading and writing is done via NSData, et. al.).
  Check to see if files exist; create and enumerate directories; move, copy, delete files; etc.
  Just alloc/init an instance and start performing operations. Thread safe.

```
NSFileManager *manager = [[NSFileManager alloc] init];

- (BOOL)createDirectoryAtPath:(NSString *)path
  withIntermediateDirectories:(BOOL)createIntermediates
                   attributes:(NSDictionary *)attributes  // permissions, etc.
                        error:(NSError **)error;

- (BOOL)isReadableFileAtPath:(NSString *)path;

- (NSArray *)contentsOfDirectoryAtPath:(NSString *)path error:(NSError **)error;
```

  Has a delegate with lots of "should" methods (to do an operation or proceed after an error).

  And plenty more. Check out the documentation.

File System

- NSString

  Path construction methods and reading/writing strings to files.

```
- (NSString *)stringByAppendingPathComponent:(NSString *)component;

- (NSString *)stringByDeletingLastPathComponent;

- (BOOL)writeToFile:(NSString *)path
         atomically:(BOOL)flag
           encoding:(NSStringEncoding)encoding  // e.g. ASCII, ISOLatin1, etc.
              error:(NSError **)error;

- (NSString *)stringWithContentsOfFile:(NSString *)path
                          usedEncoding:(NSStringEncoding *)encoding
                                 error:(NSError **)error;
```

  And plenty more. Check out the documentation.

SQLite

- SQL in a single file

  Fast, low memory, reliable.
  Open Source, comes bundled in iOS.
  Not good for everything (e.g. video or even serious sounds/images).
  Not a server-based technology (not great at concurrency, but usually not a big deal on a phone).

- API

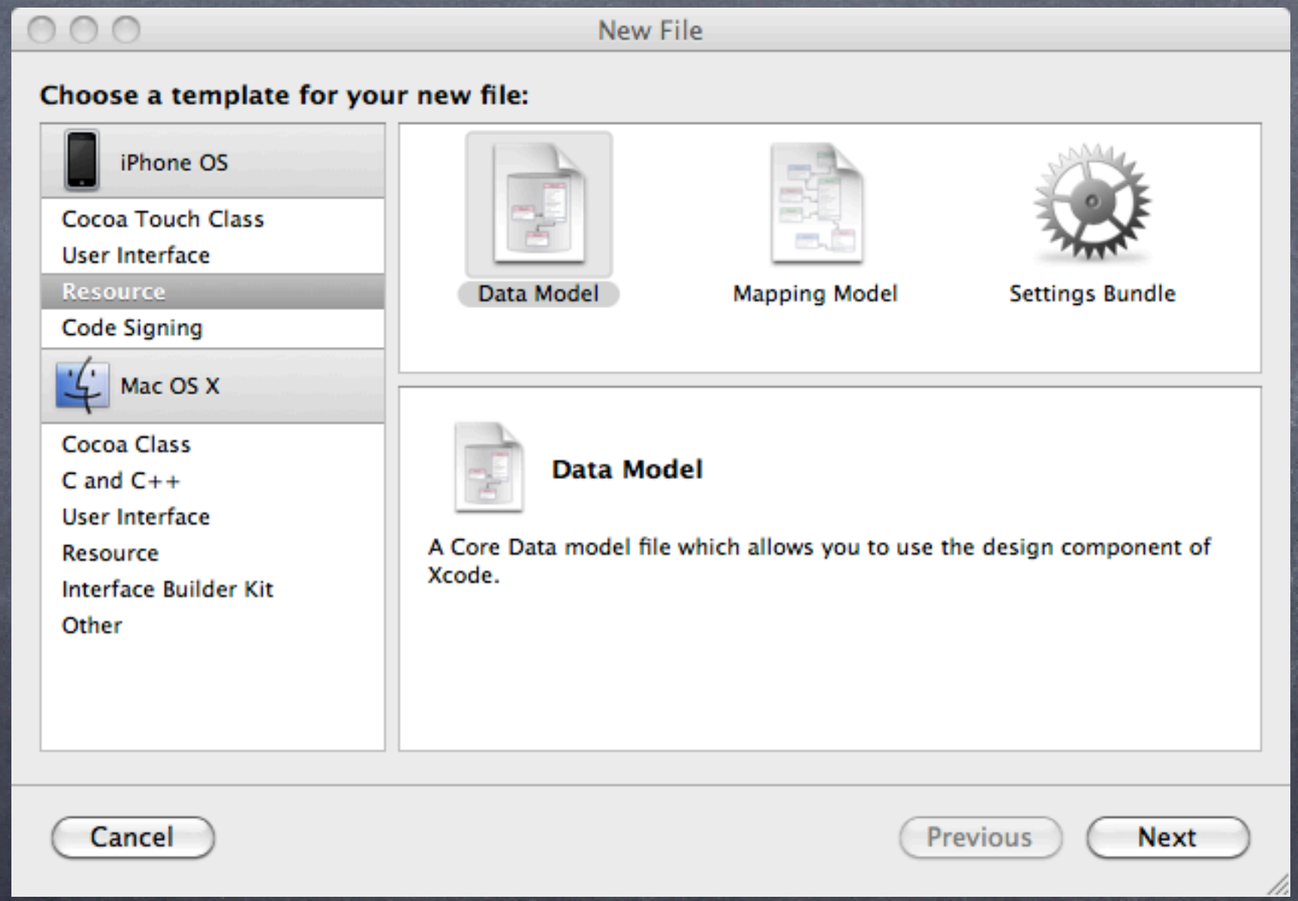
```
int sqlite3_open(const char *filename, sqlite3 **db);   // get a database into db

int sqlite3_exec(sqlite3 *db,                           // execute SQL statements
                 const char *sql,
                 int (*callback)(void *, int, char **, char **),
                 void *context,
                 char **error);

int mycallback(void *context, int count, char **values, char **cols);   // data returned

int sqlite3_close(sqlite3 *db);                         // close the database
```

Core Data

- But we're object-oriented programmers and we don't like C APIs!
- We want to store our data using object-oriented programming!
- Enter Core Data
- It's a way of creating an object graph backed by a database
  Usually SQL.
- How does it work?
  Create a visual mapping (using Xcode tool) between database and objects.
  Create and query for objects using object-oriented API.
  Access the "columns in the database table" using `@property`s on those objects.

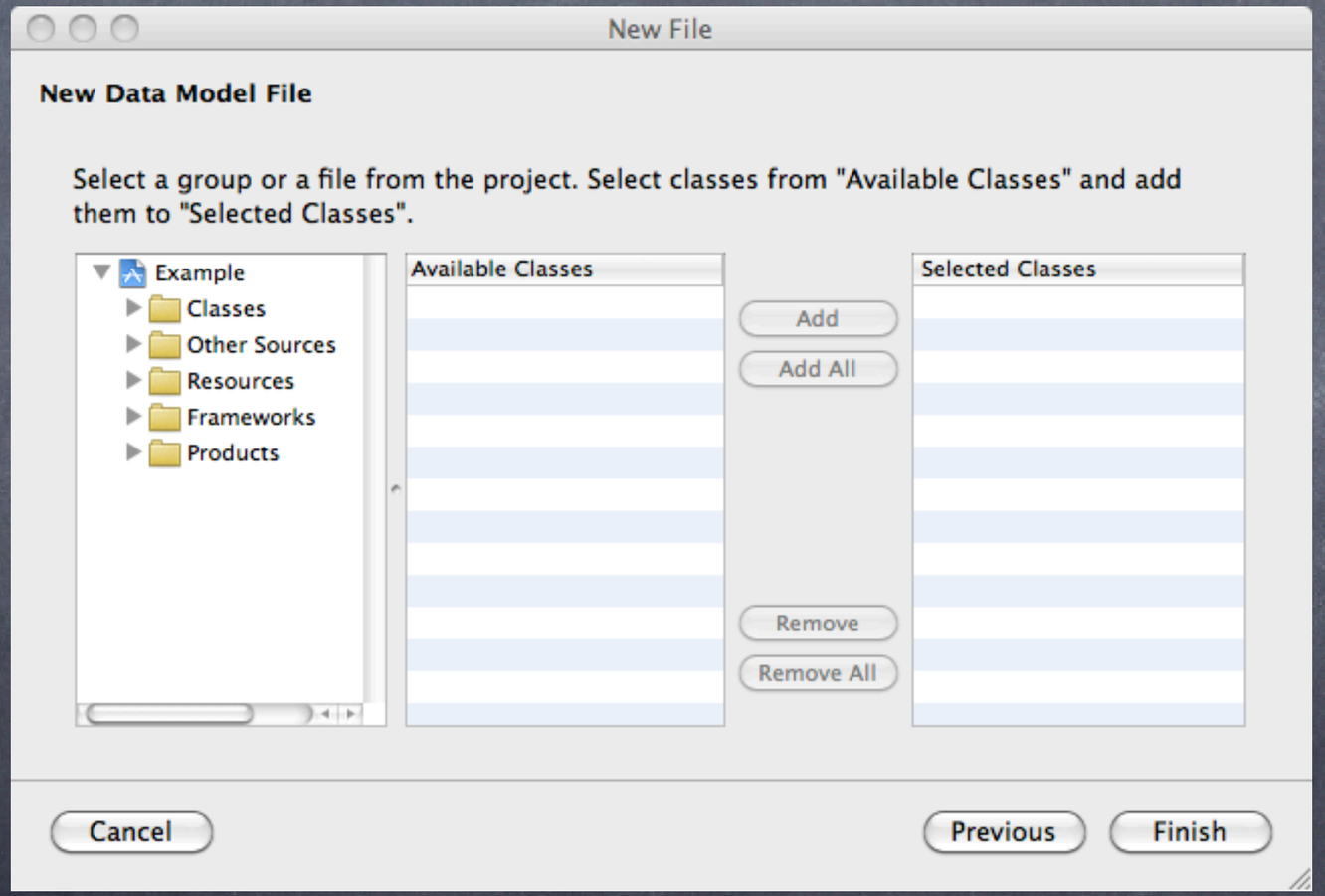
Core Data

- Creating the data mapping (data model) graphically
  - Start with New File … in Xcode
  - Under Resource, Choose Data Model



Core Data

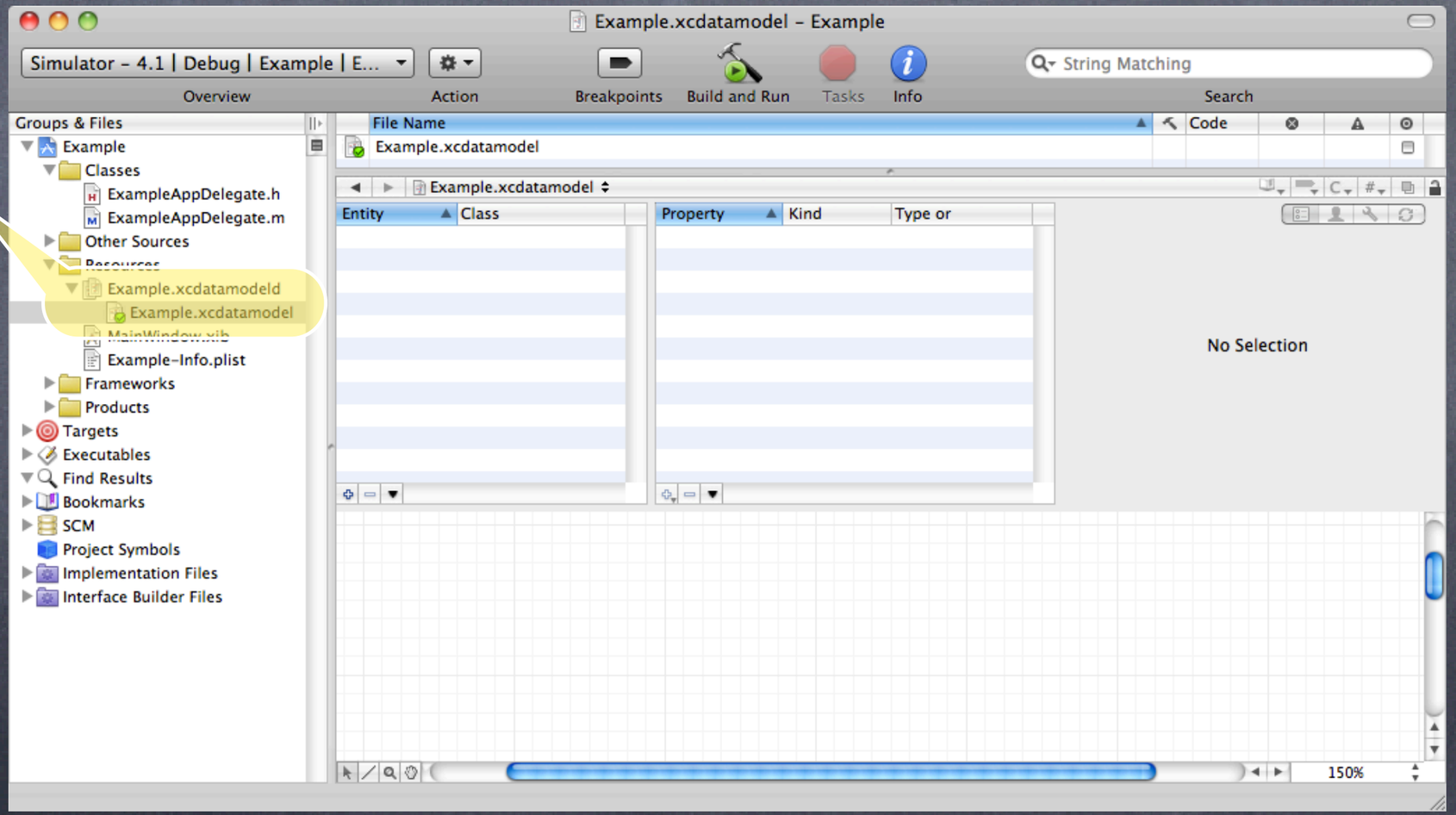
- You could use an existing class and its properties as a template
  But we almost never do. Usually we start with the mapping and generate the objects from it.
  So we almost always just skip this window when it comes up (click Finish without doing anything).



Core Data

- This creates a `.xcdatamodel` file
  Think of it like a `.xib` for our data mapping between the database and our objects

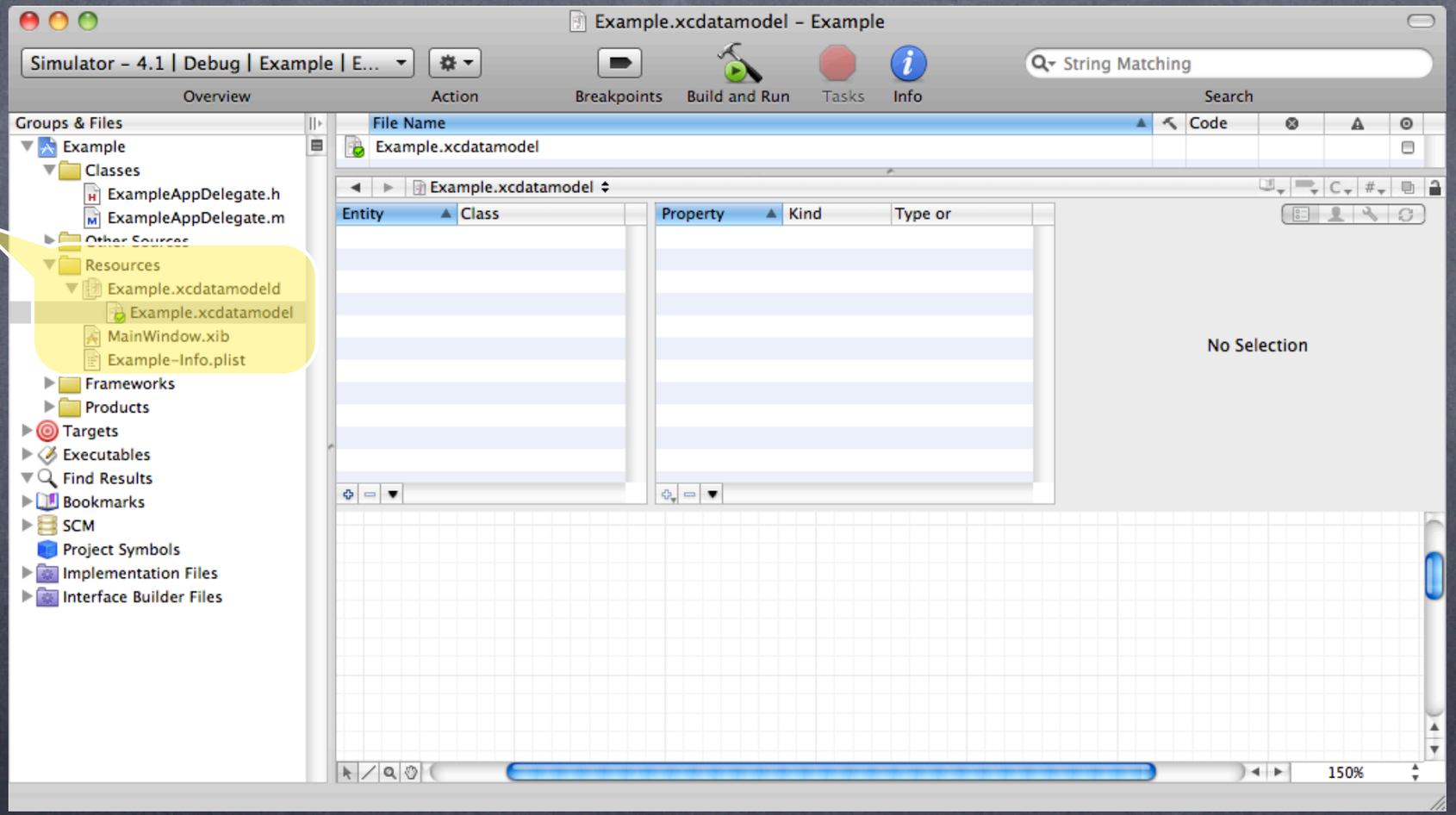
Note that it put our `.xcdatamodel` file into a `.xcdatamodeld` directory. This is because we can have multiple versions of our `.xcdatamodel` and support backward compatibility if we want.



Core Data

- This creates a `.xcdatamodel` file
  Think of it like a `.xib` for our data mapping between the database and our objects

Usually we drag the `.xcdatamodeld` directory into our Resources folder.



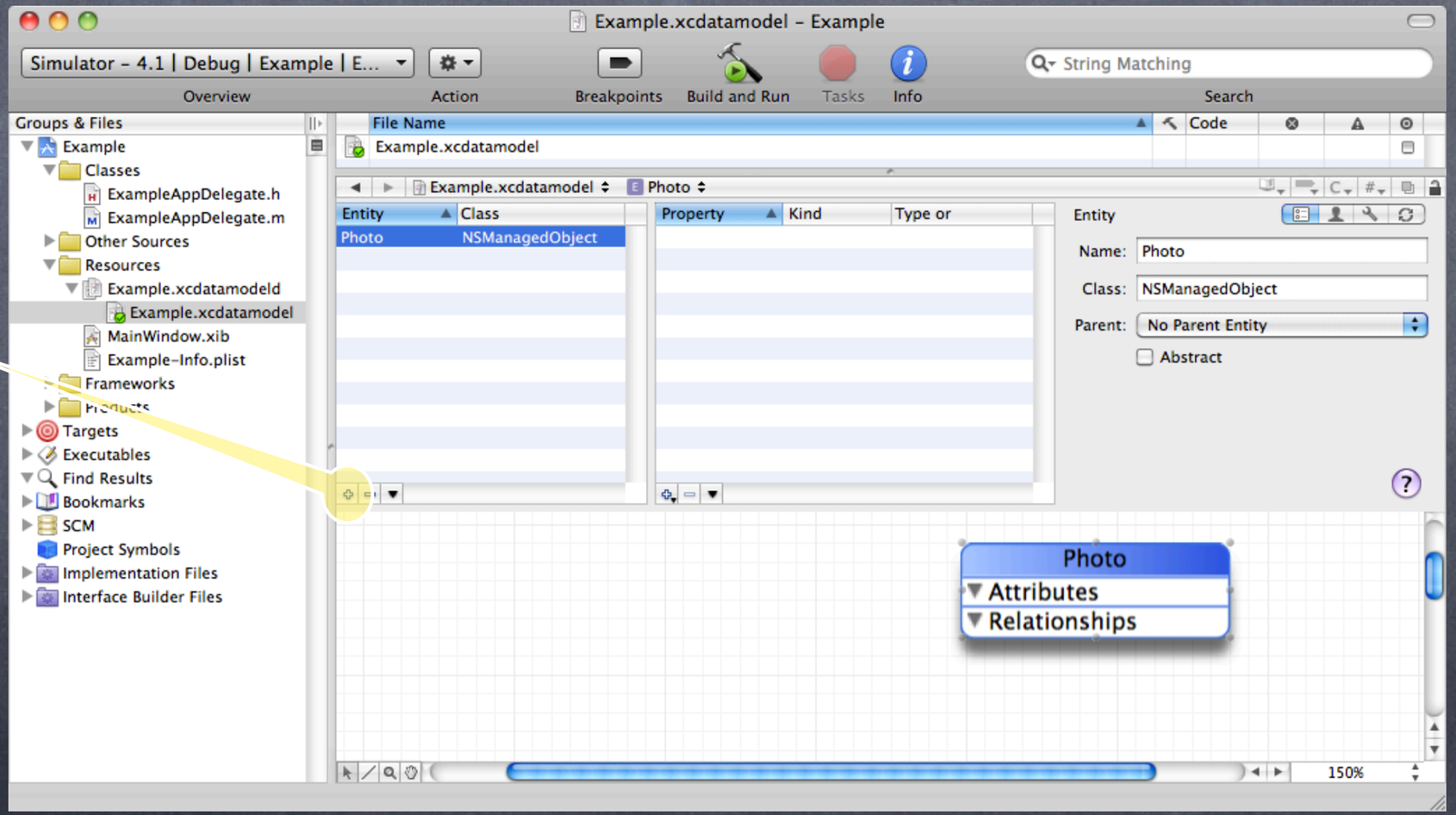
Core Data

- First we will create an Entity

  An Entity maps to a "table" in the database and maps to an "object" in our code.

  This Photo Entity is intended to represent a photograph (maybe one downloaded from Flickr?).

Click here to add an Entity.

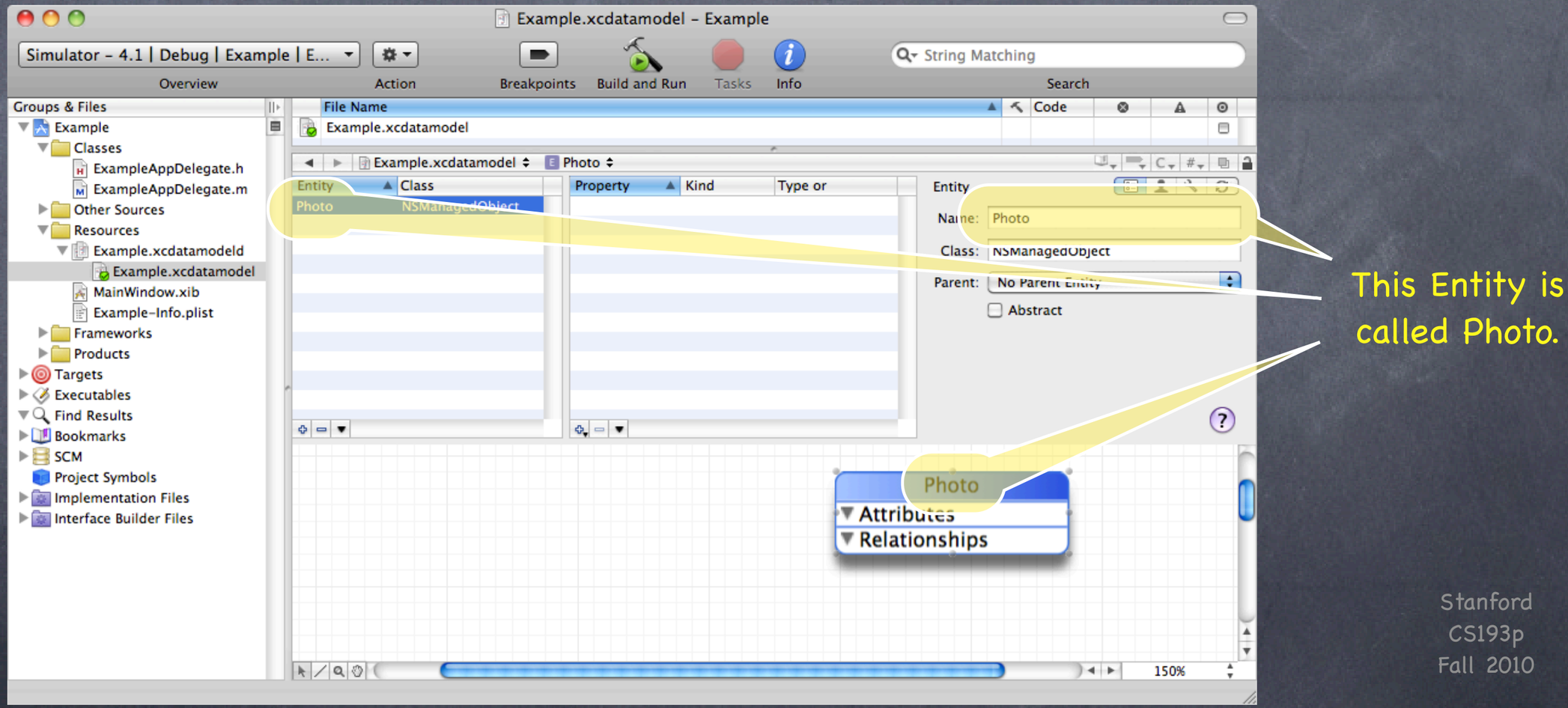


Core Data

- First we will create an Entity

  An Entity maps to a "table" in the database and maps to an "object" in our code.

  This Photo Entity is intended to represent a photograph (maybe one downloaded from Flickr?).

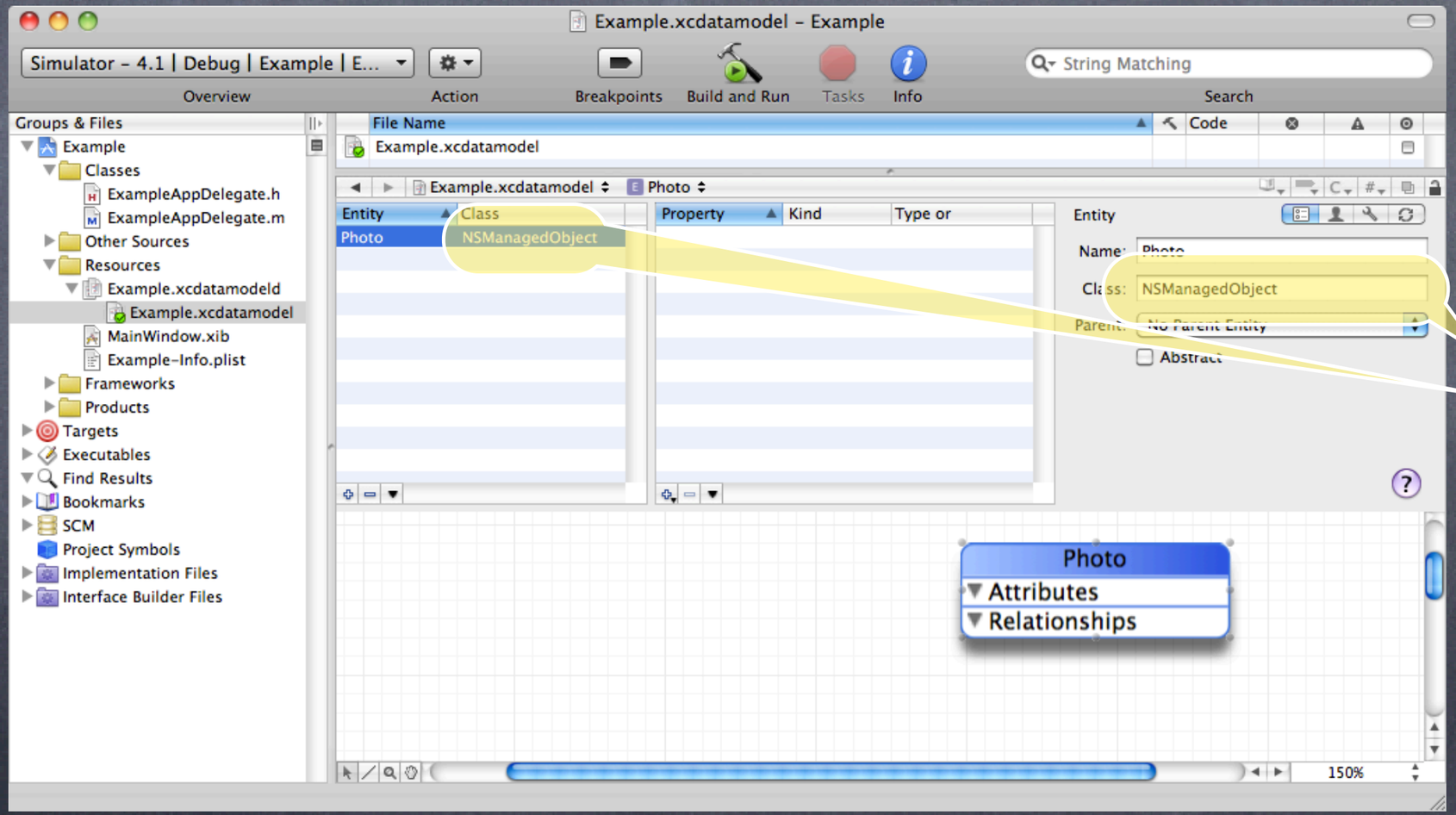


Core Data

- First we will create an Entity

  An Entity maps to a "table" in the database and maps to an "object" in our code.

  This Photo Entity is intended to represent a photograph (maybe one downloaded from Flickr?).



It will appear in our code as an `NSManagedObject` (for now).

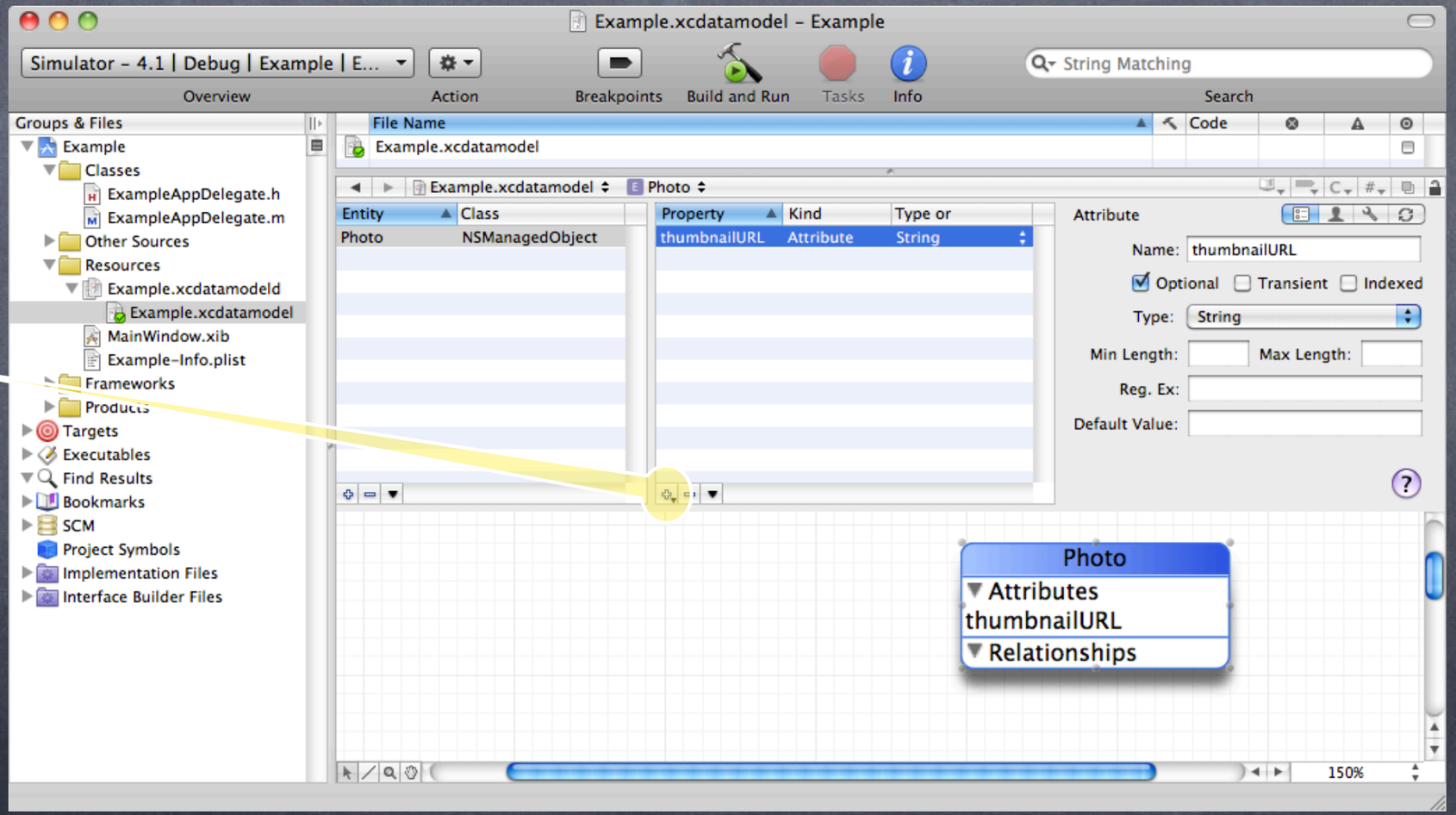
Core Data

- Then we add Attributes to our Entity

  An Attribute maps to a "column" in the database and maps to a property in our objects.

  A thumbnailURL stores the URL to use to download a thumbnail image of our Photo.

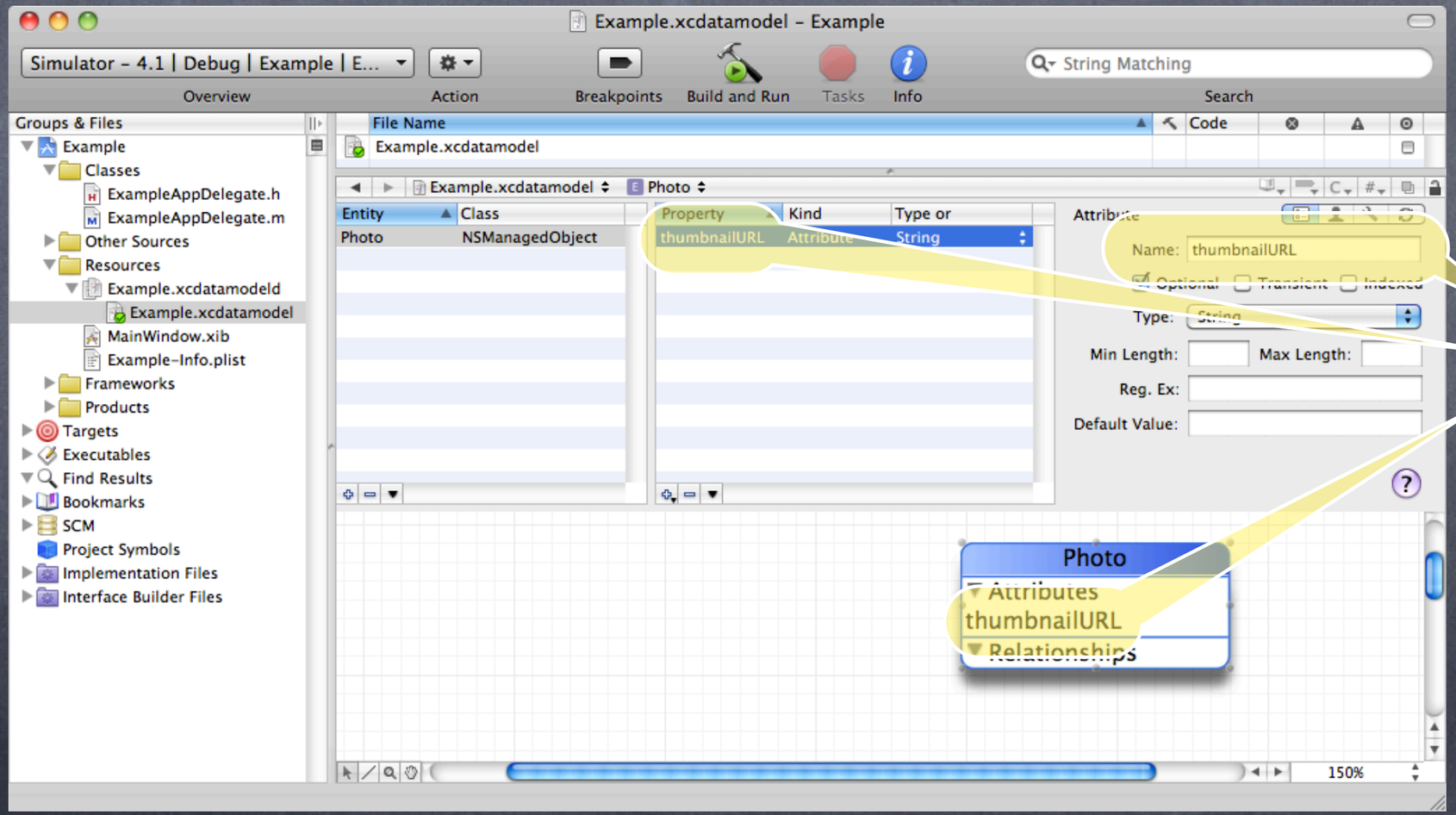
Click here to add an attribute.



Core Data

- Then we add Attributes to our Entity

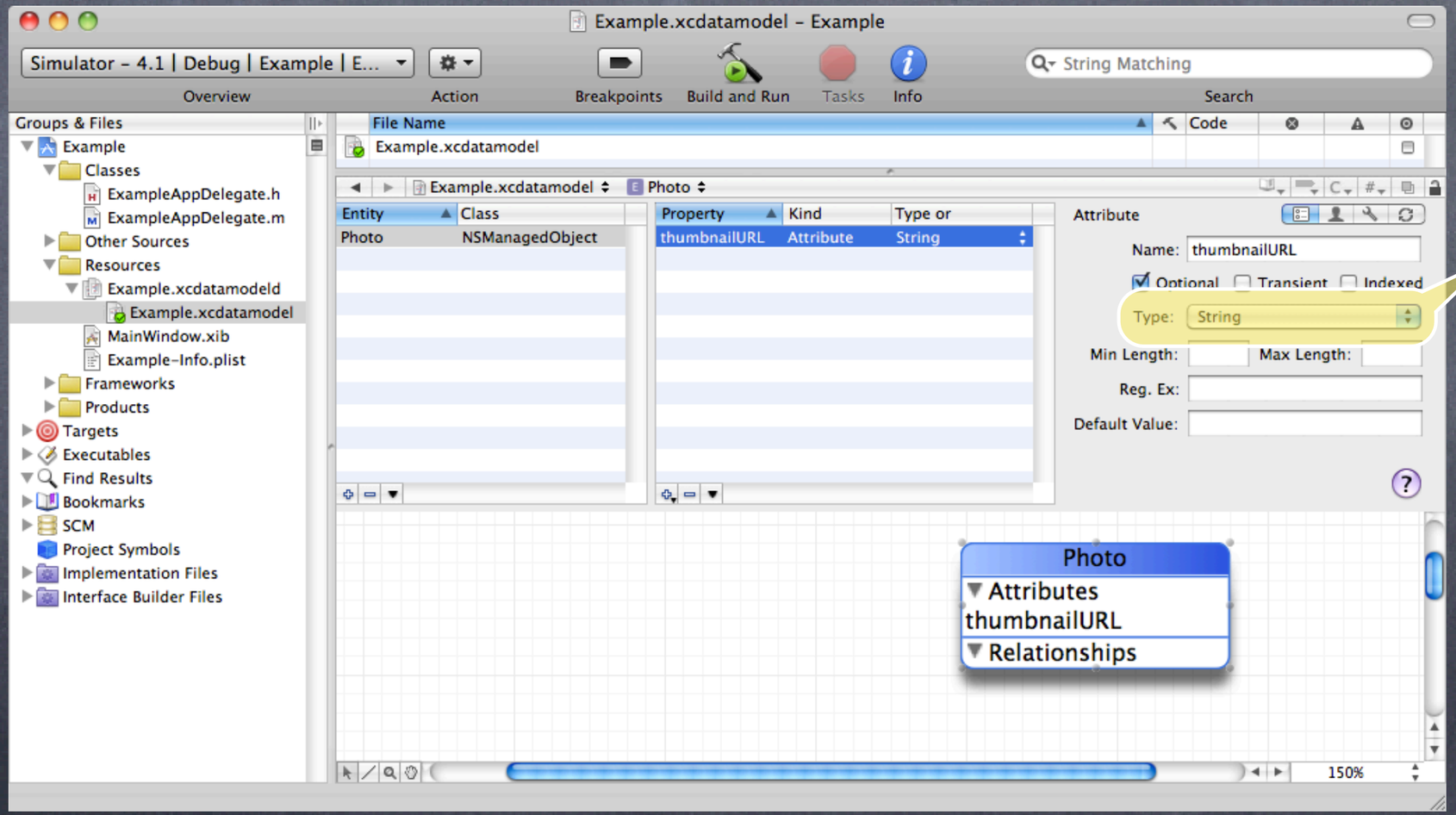
  An Attribute maps to a "column" in the database and maps to a property in our objects.
  A thumbnailURL stores the URL to use to download a thumbnail image of our Photo.



Core Data

- Then we add Attributes to our Entity

  An Attribute maps to a "column" in the database and maps to a property in our objects.
  A thumbnailURL stores the URL to use to download a thumbnail image of our Photo.

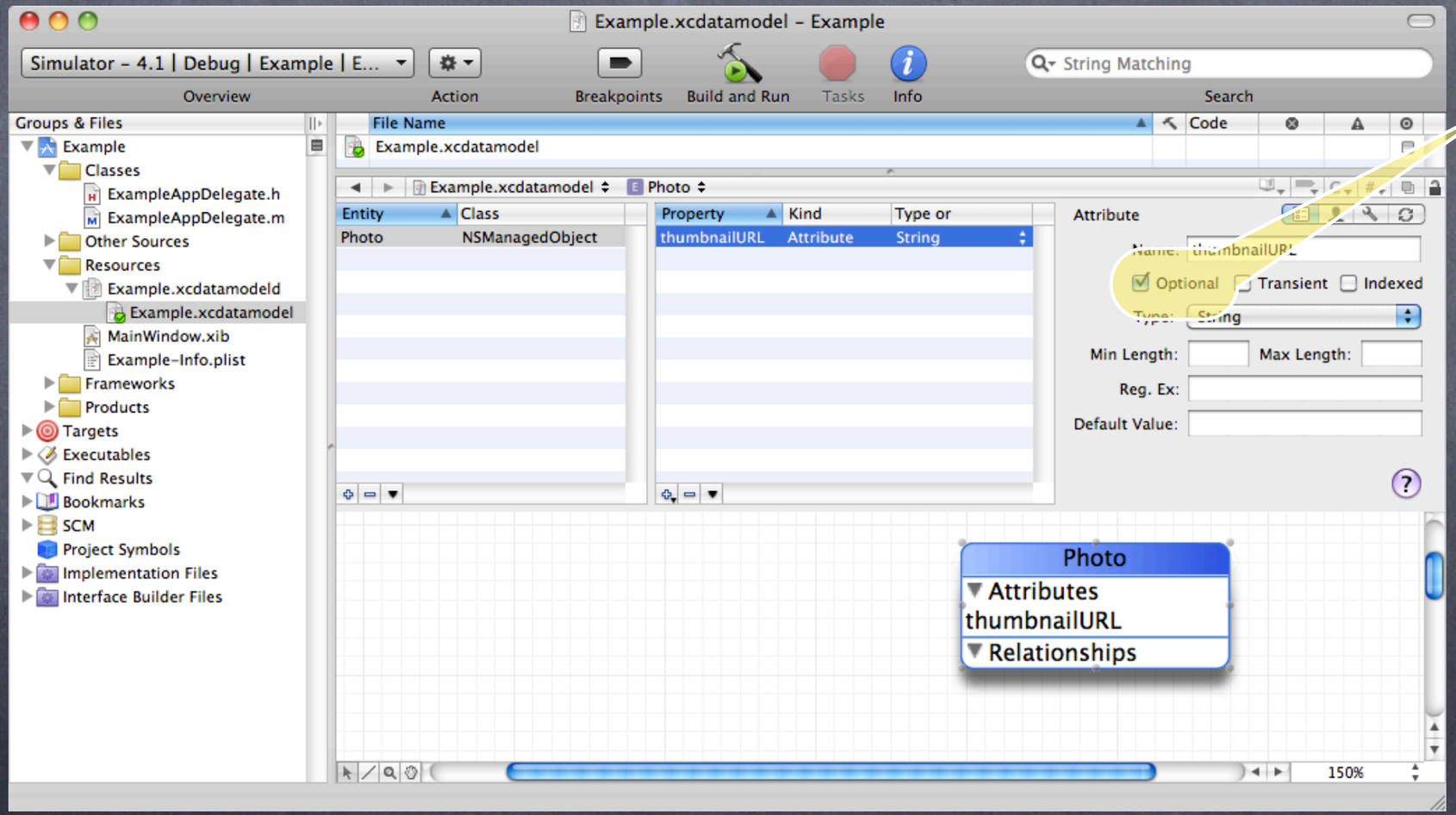


The type of this attribute is "String." This will map to an `NSString` in our code.

Core Data

- Then we add Attributes to our Entity

  An Attribute maps to a "column" in the database and maps to a property in our objects.
  A thumbnailURL stores the URL to use to download a thumbnail image of our Photo.

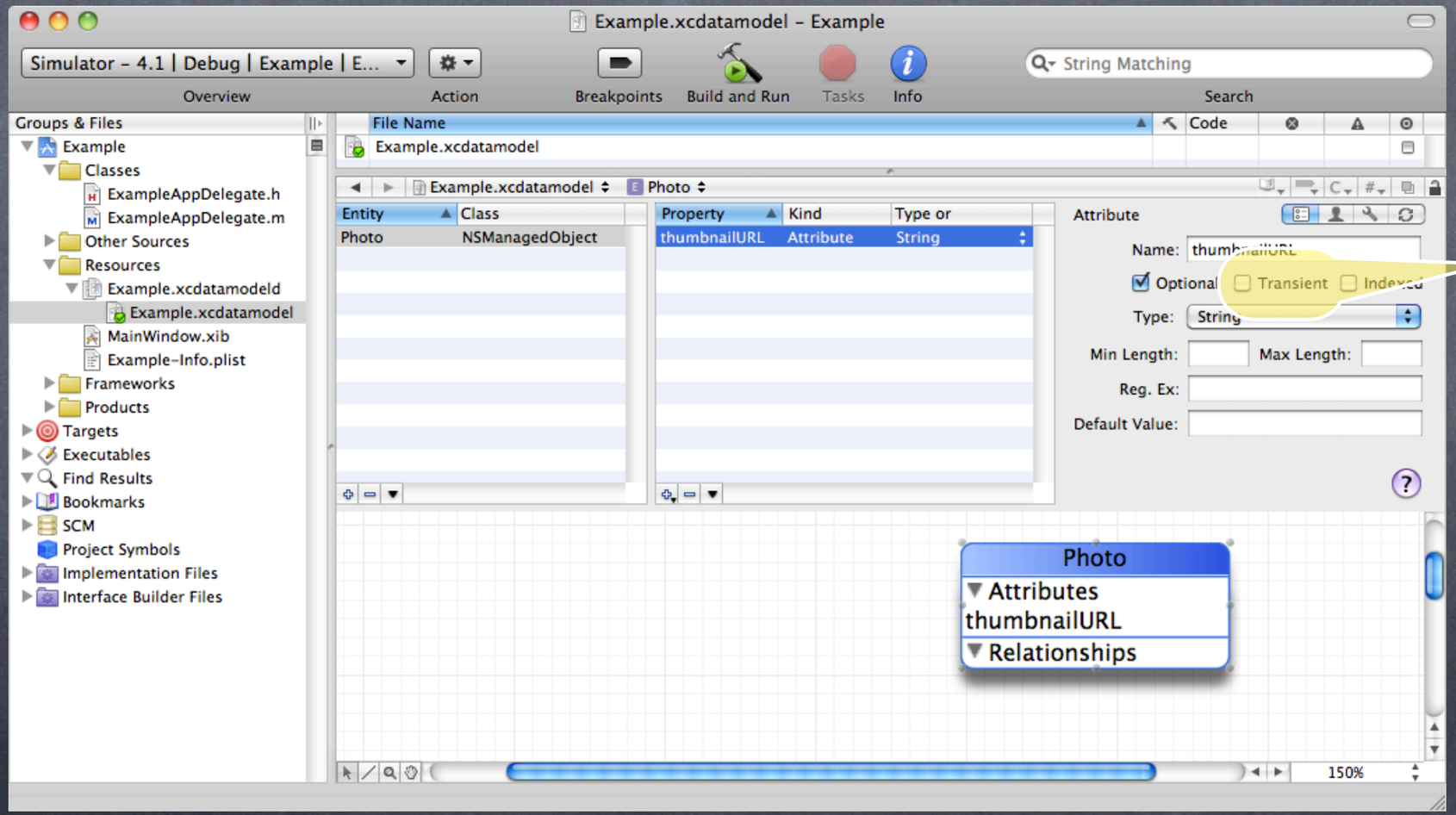


Optional
Objects are allowed to be created without specifying this attribute's value.

Core Data

- Then we add Attributes to our Entity

  An Attribute maps to a "column" in the database and maps to a property in our objects.
  A thumbnailURL stores the URL to use to download a thumbnail image of our Photo.

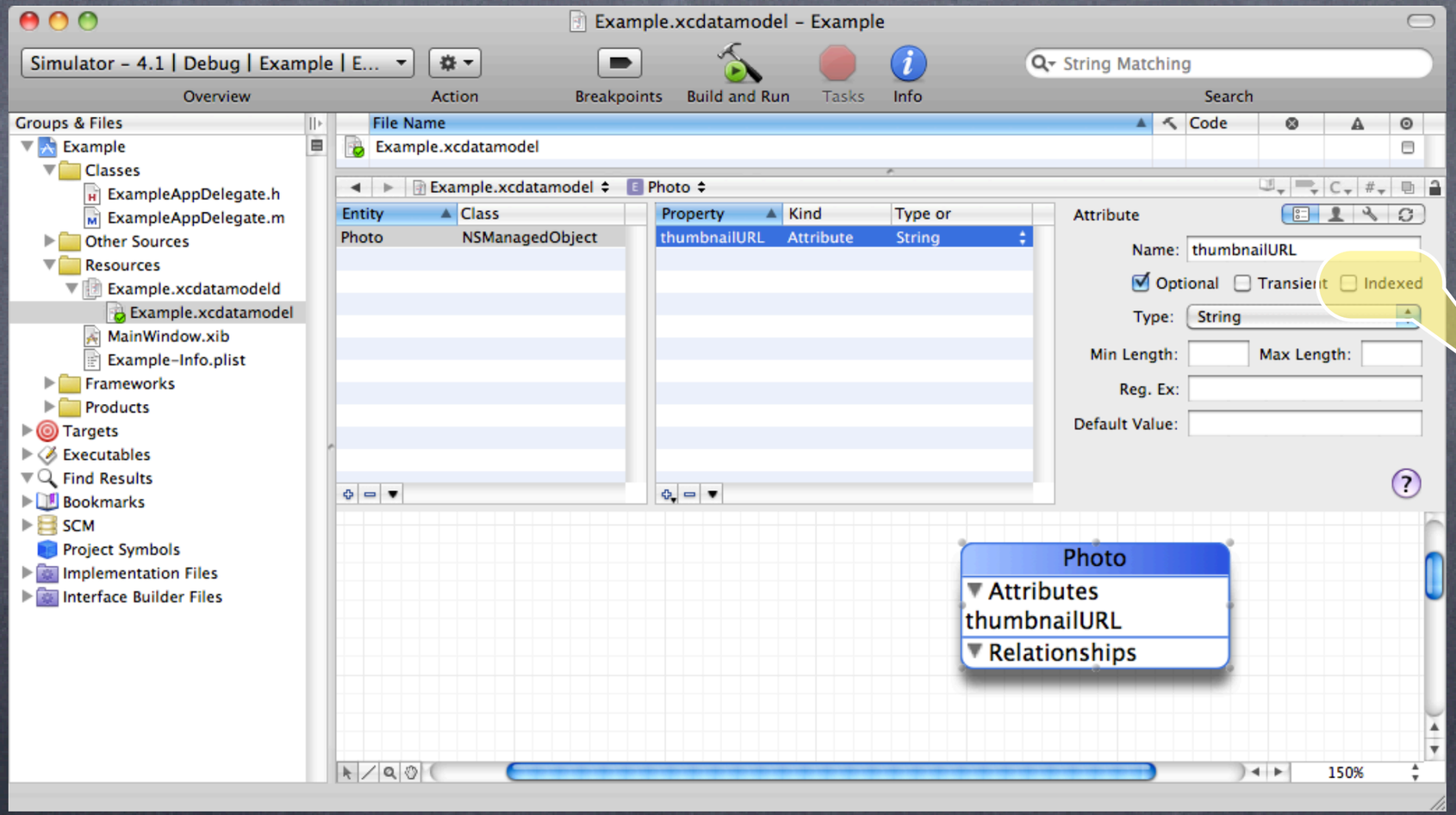


Transient
This attribute is on the object-side only (not stored in the database).

Core Data

- Then we add Attributes to our Entity

  An Attribute maps to a "column" in the database and maps to a property in our objects.
  A thumbnailURL stores the URL to use to download a thumbnail image of our Photo.



Indexed

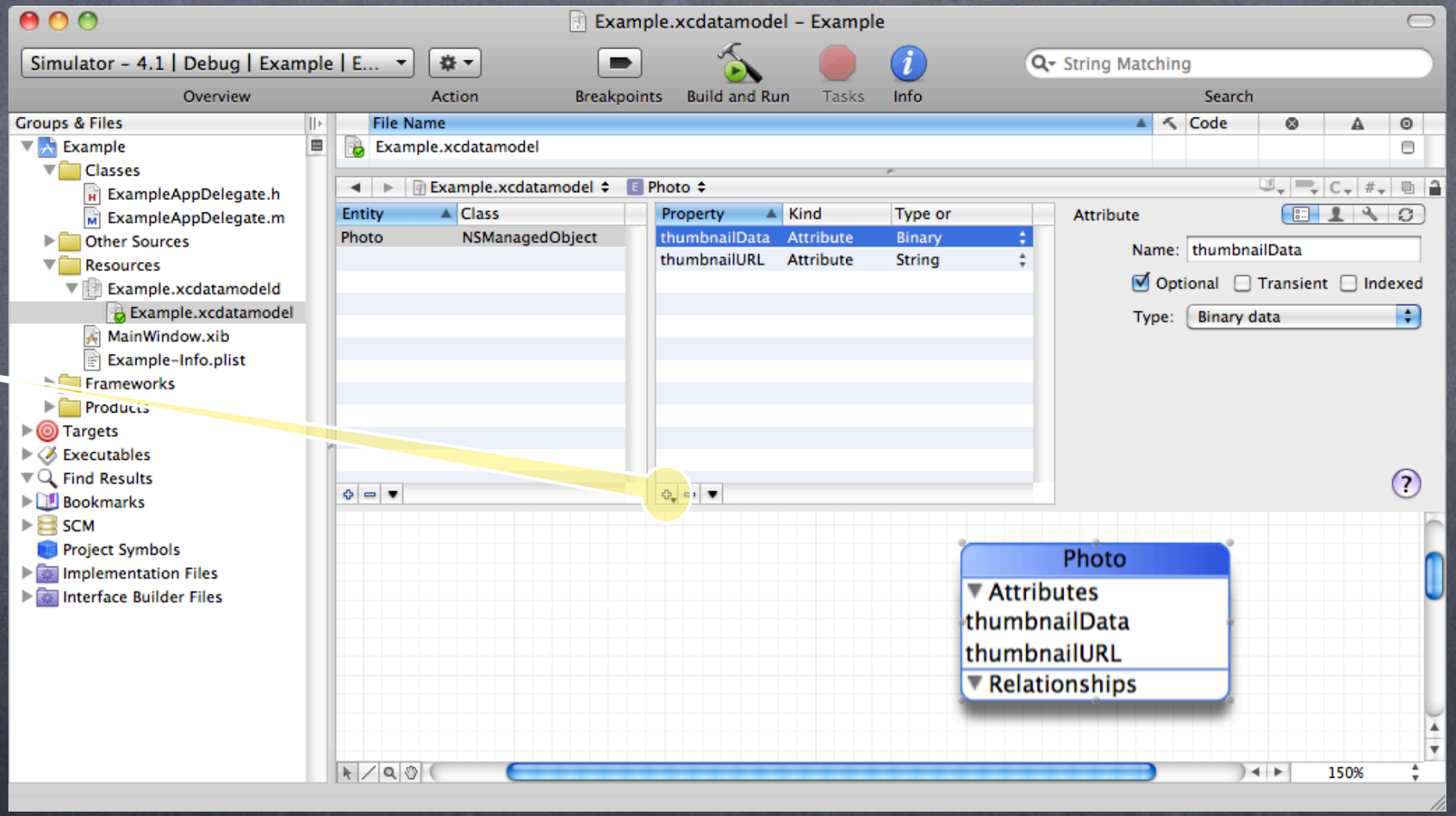
Make searches on this attribute efficient.

Core Data

- Let's add another Attribute

  A thumbnailData is an Attribute which stores the actual image data for our Photo's thumbnail.

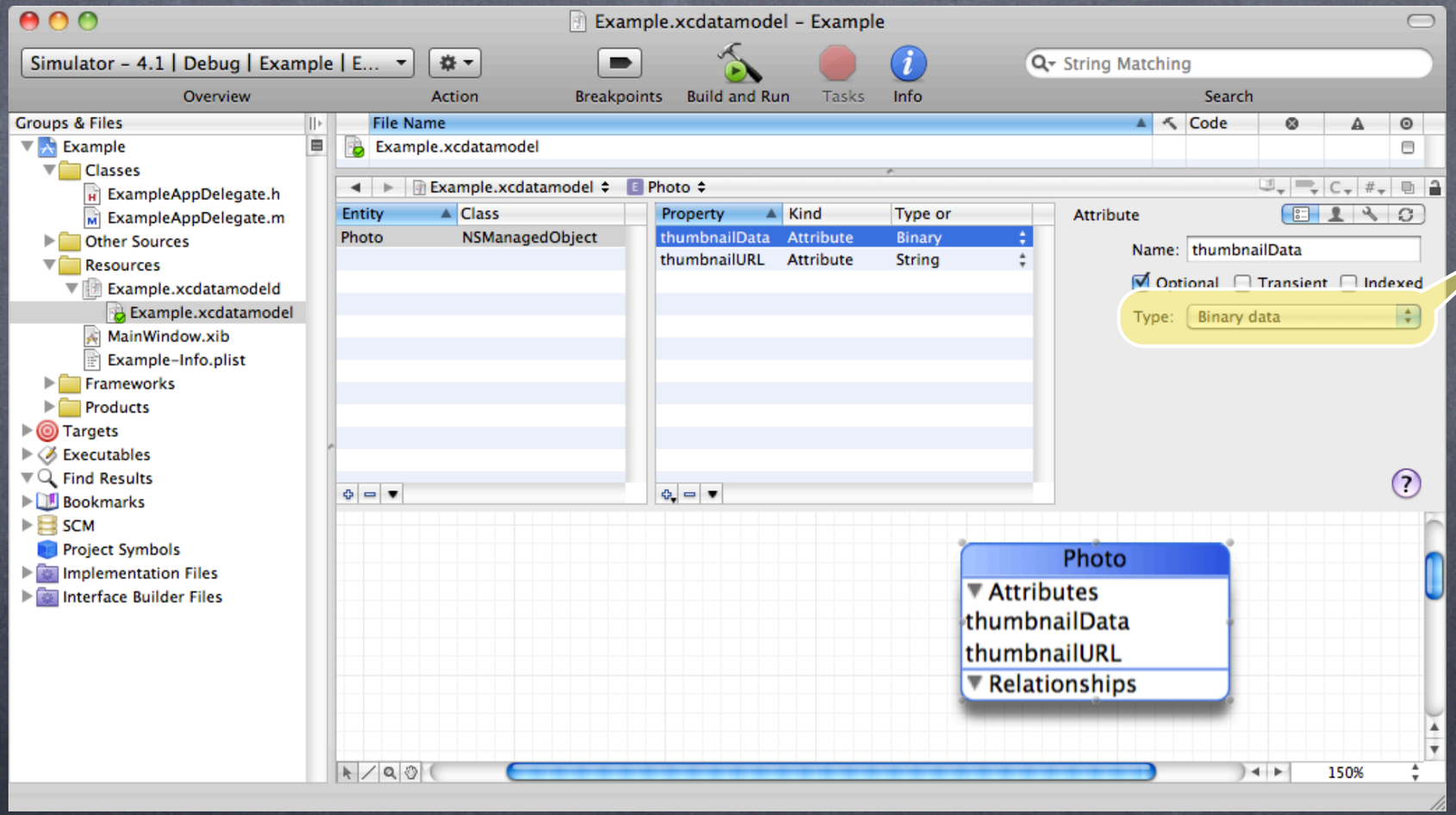
Click here to add another attribute.



Core Data

- Let's add another Attribute

  A thumbnailData is an Attribute which stores the actual image data for our Photo's thumbnail.

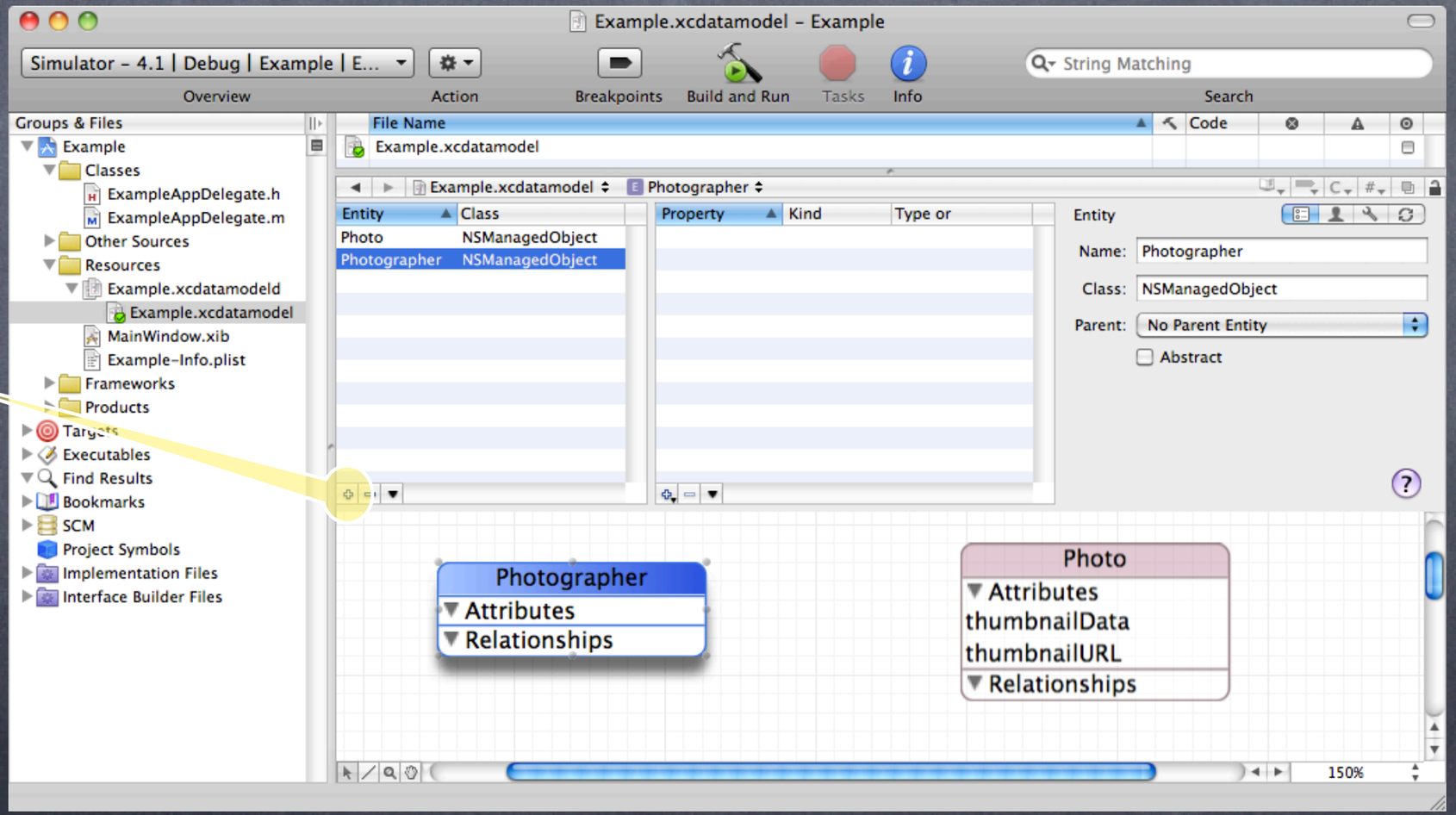


The type of this attribute is "Binary data." This will map to an NSData in our code.

Core Data

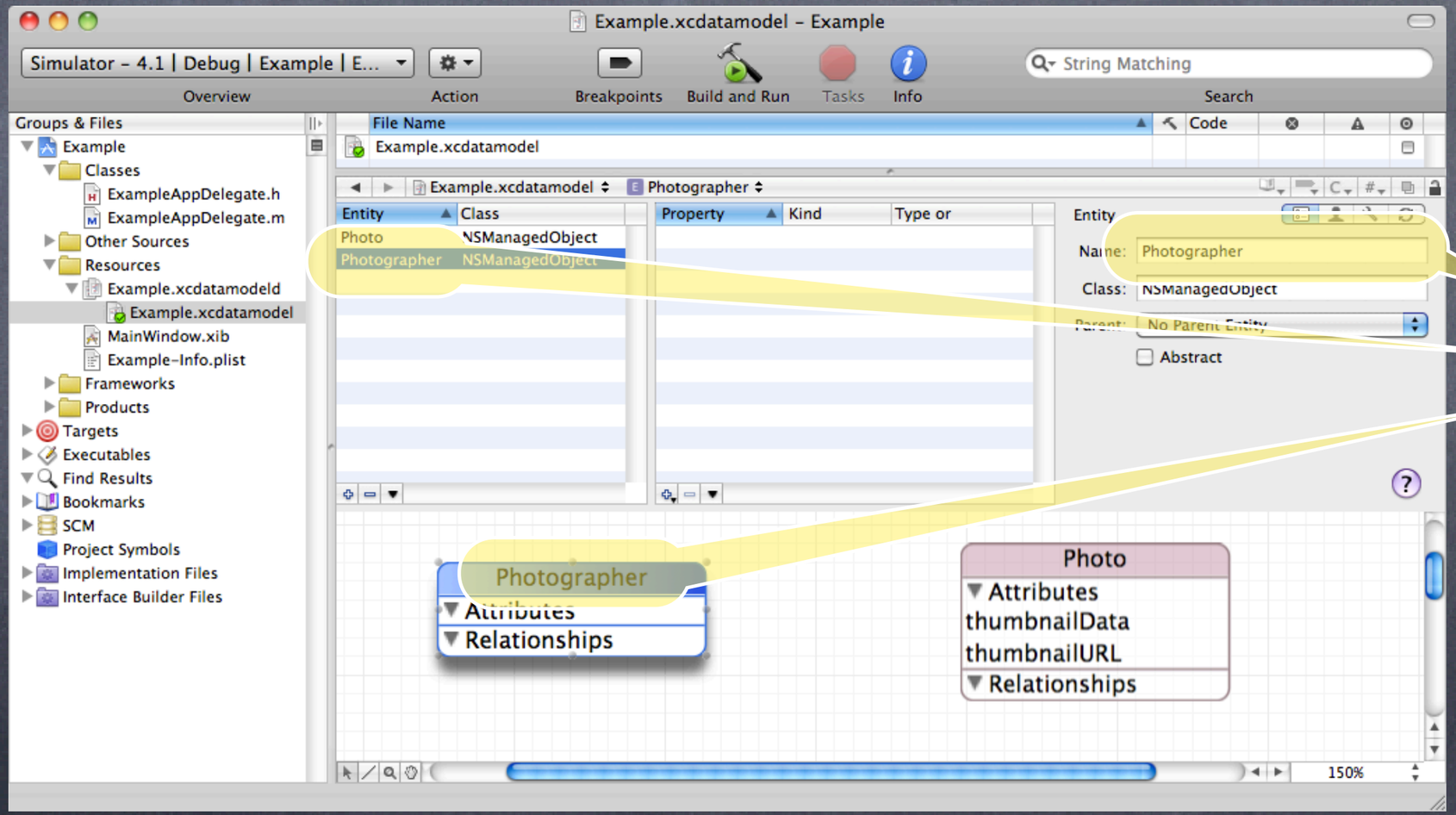
- Let's create another Entity
  A Photographer is an Entity representing a person who takes photos.

Click here to add another Entity.



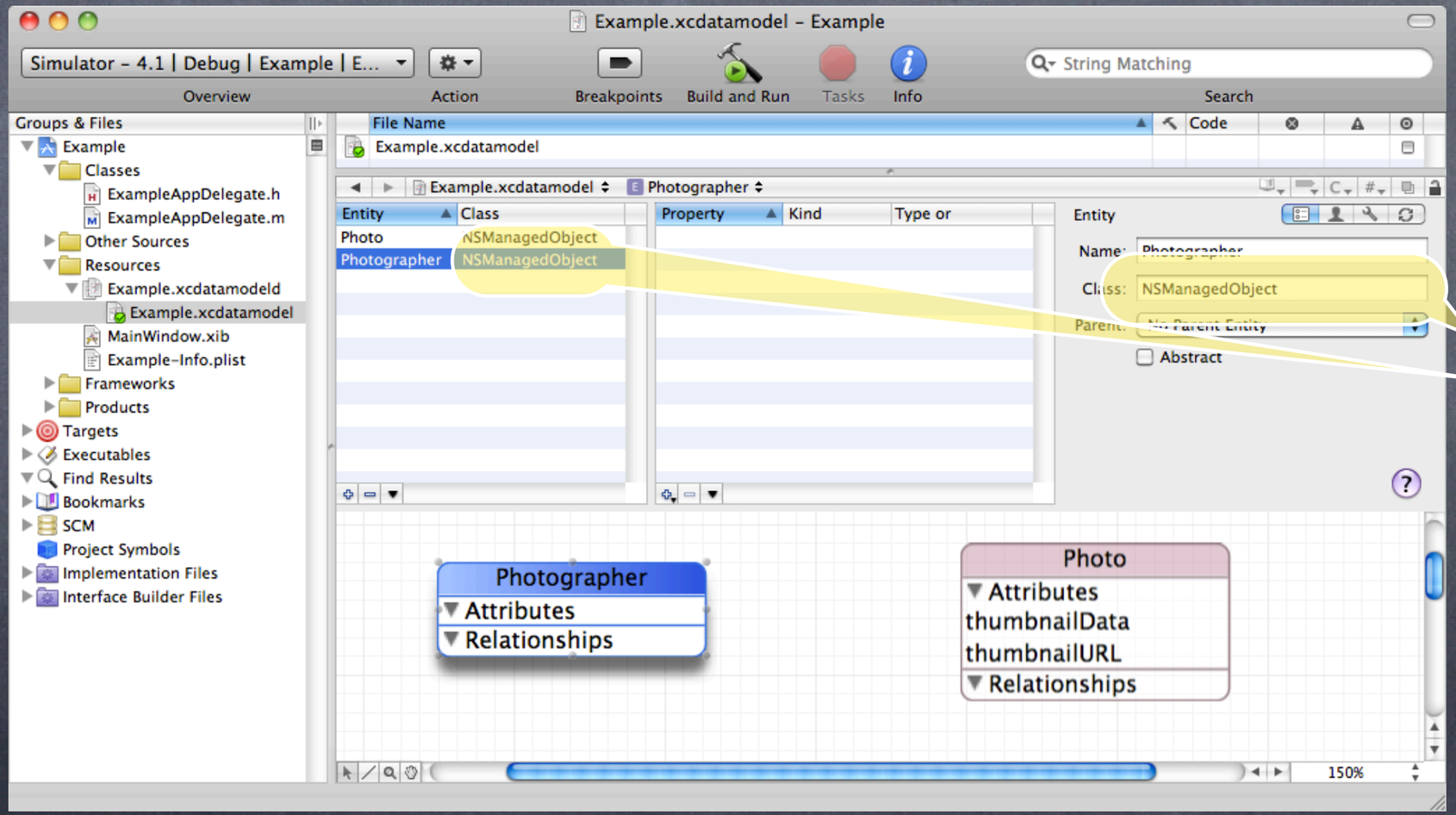
Core Data

- Let's create another Entity
  A Photographer is an Entity representing a person who takes photos.



Core Data

- Let's create another Entity
  A Photographer is an Entity representing a person who takes photos.

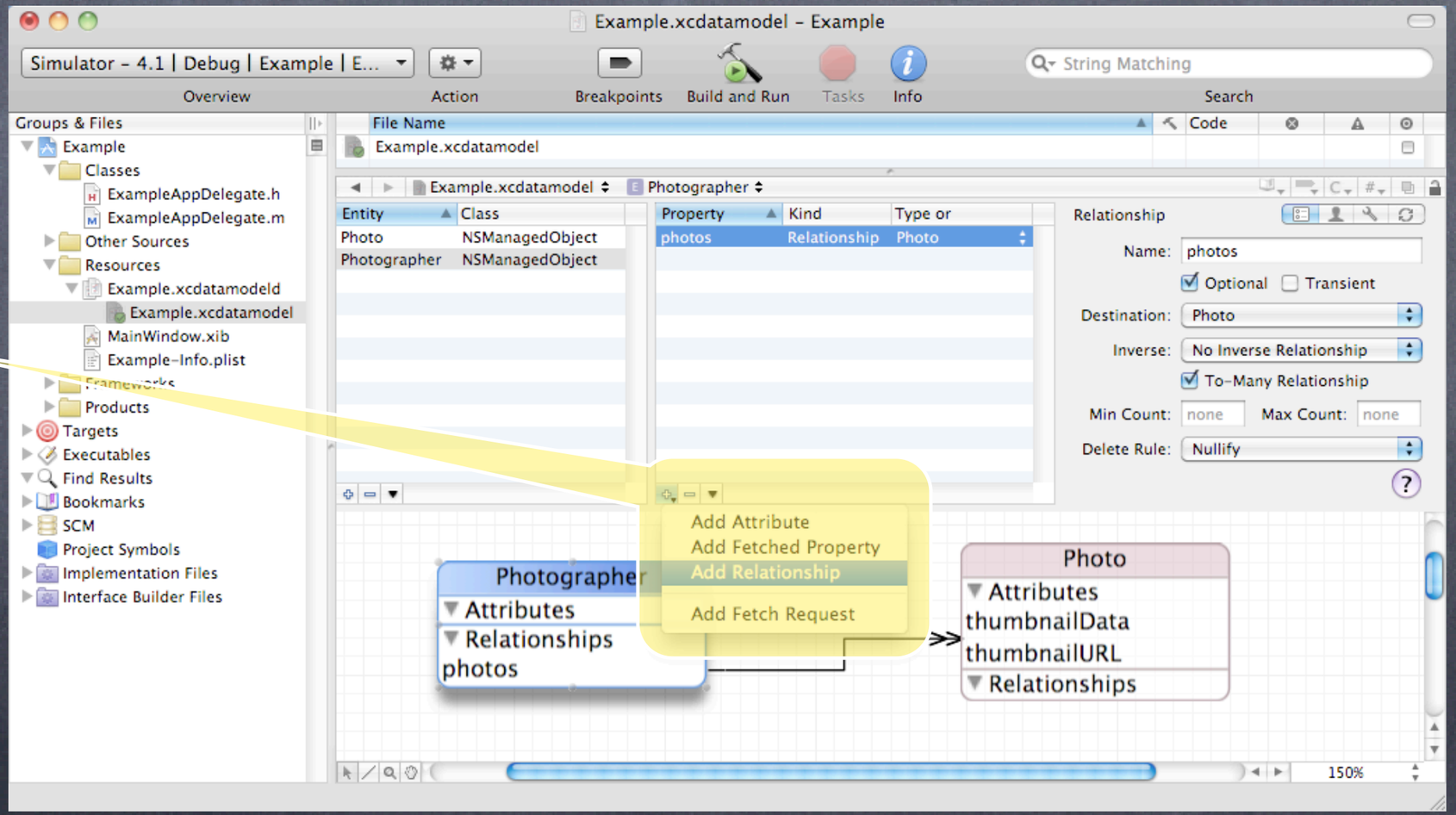


It is also an NSManagedObject (for now).

Core Data

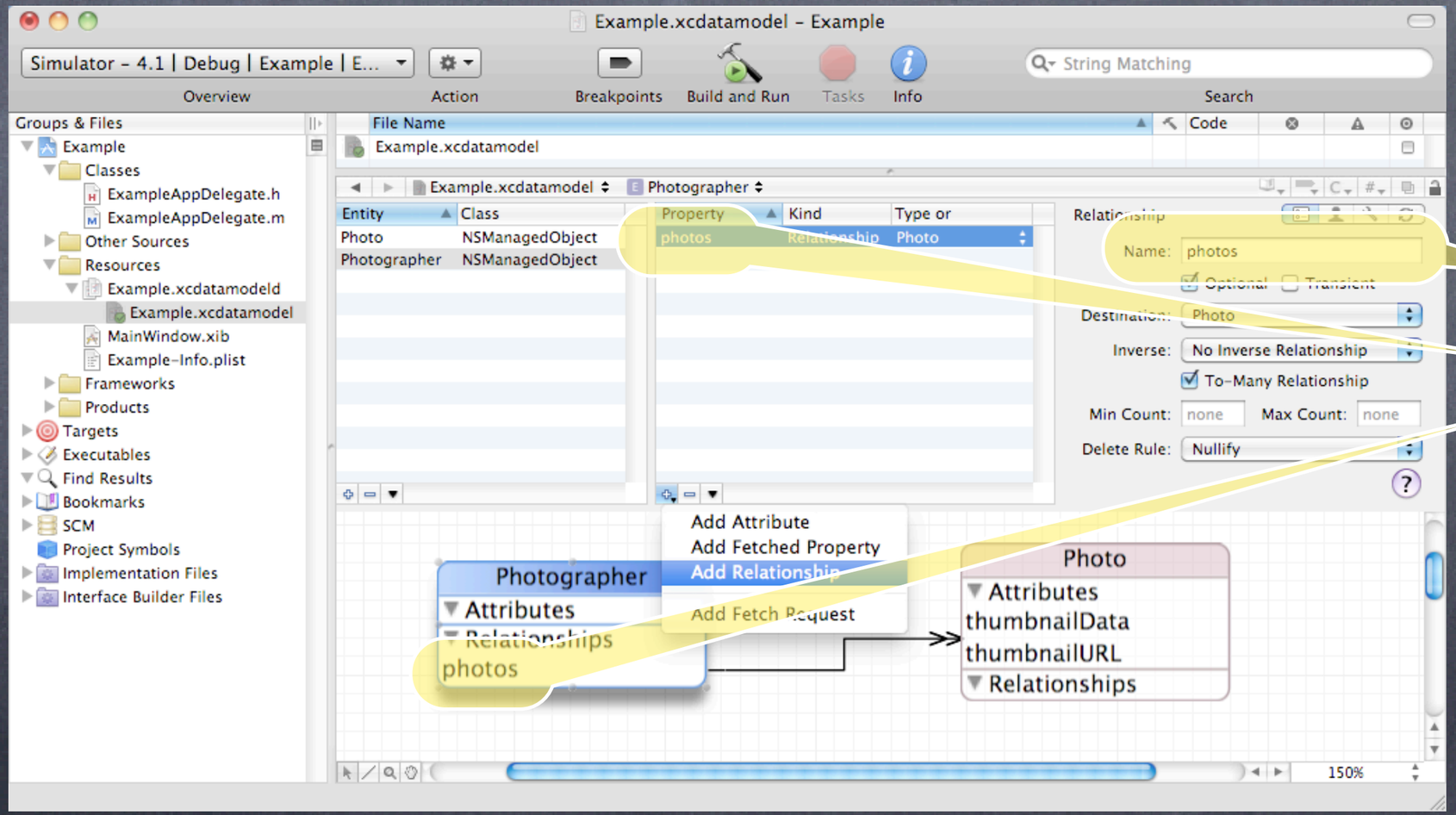
- Now we will add a relationship between one Entity and another
  This is a "to-many" Relationship called photos (maps to an (NSSet *) property in our object).
  In other words, Photographer objects will have an NSSet of photos (taken by that photographer).

Here's where we add a Relationship.



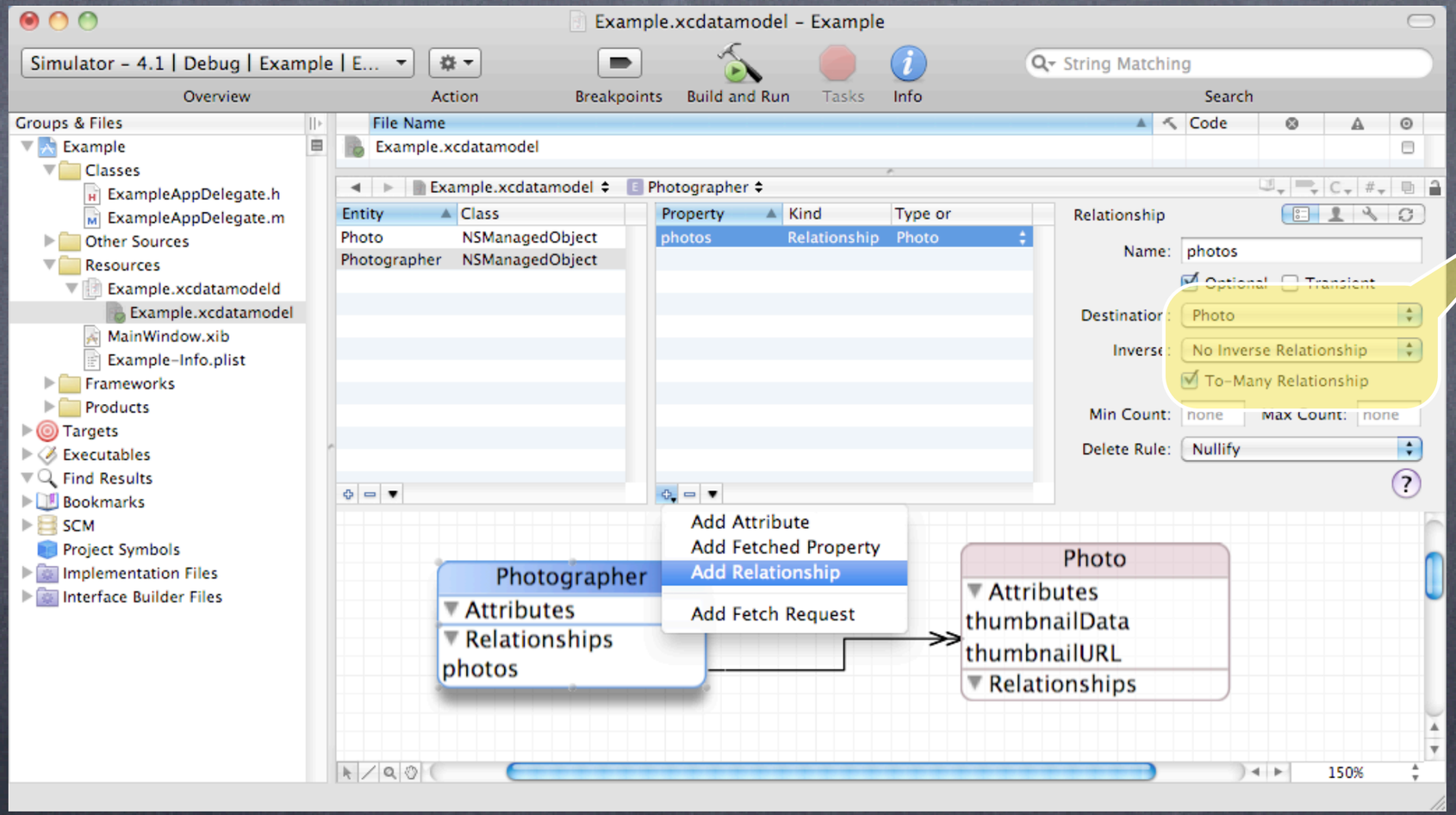
Core Data

- Now we will add a relationship between one Entity and another
  This is a "to-many" Relationship called photos (maps to an (NSSet *) property in our object).
  In other words, Photographer objects will have an NSSet of photos (taken by that photographer).



Core Data

- Now we will add a relationship between one Entity and another
  This is a "to-many" Relationship called photos (maps to an (NSSet *) property in our object).
  In other words, Photographer objects will have an NSSet of photos (taken by that photographer).



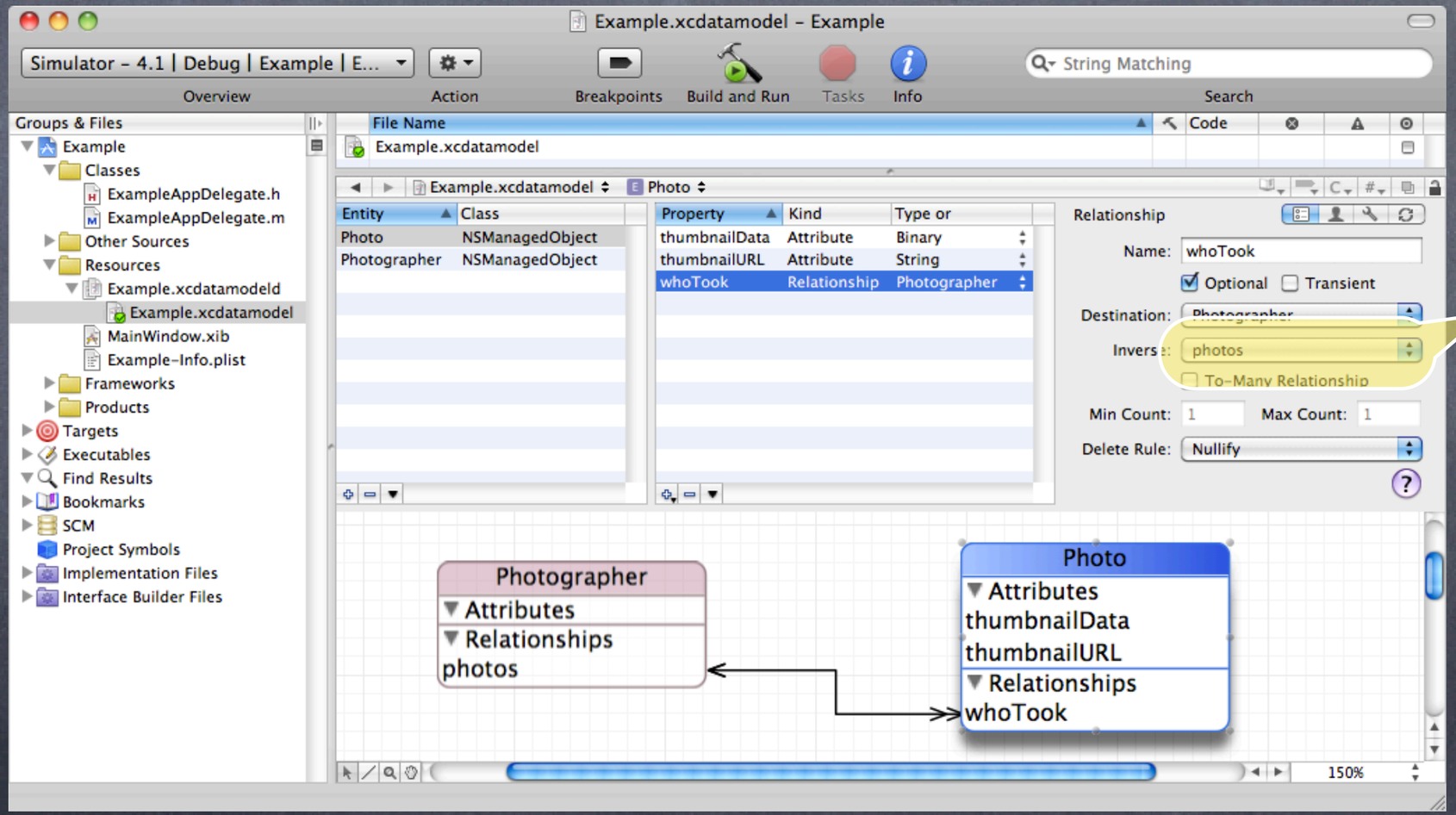
Its destination is a Photo object, it has no inverse relationship (yet), and it is a "To-Many" Relationship.

Core Data

- Relationships can go in both directions (inverse relationships)

  We've added the whoTook Relationship on Photo which points to a Photographer.

  whoTook is not a to-many relationship, so it will be an NSManagedObject property (not an NSSet).



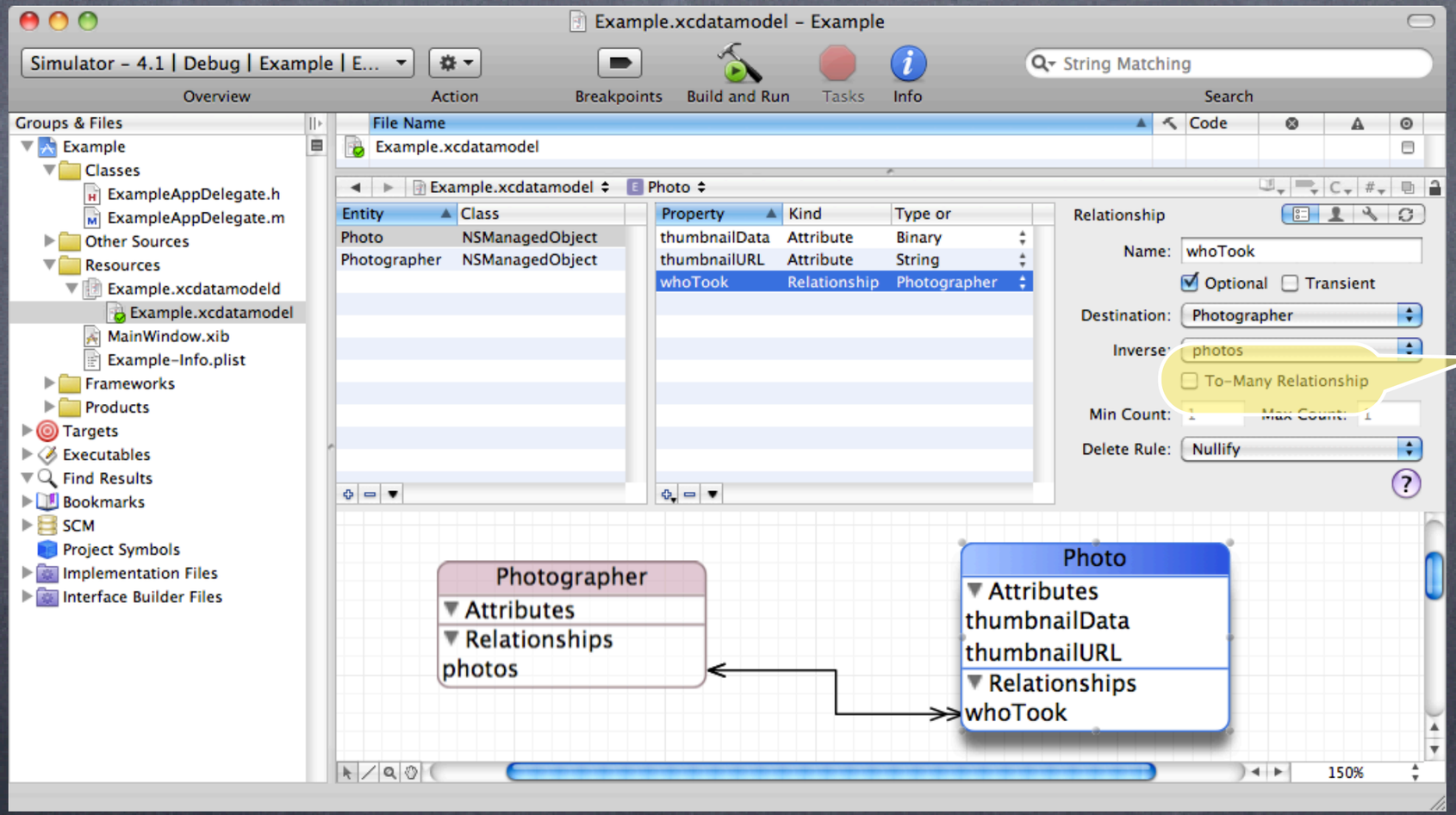
whoTook is the Inverse of the photos relationship.

Core Data

- Relationships can go in both directions (inverse relationships)

  We've added the whoTook Relationship on Photo which points to a Photographer.

  whoTook is not a to-many relationship, so it will be an NSManagedObject property (not an NSSet).



Notice that the To-Many box is not checked for the whoTook Relationship (like it is for the photos Relationship).

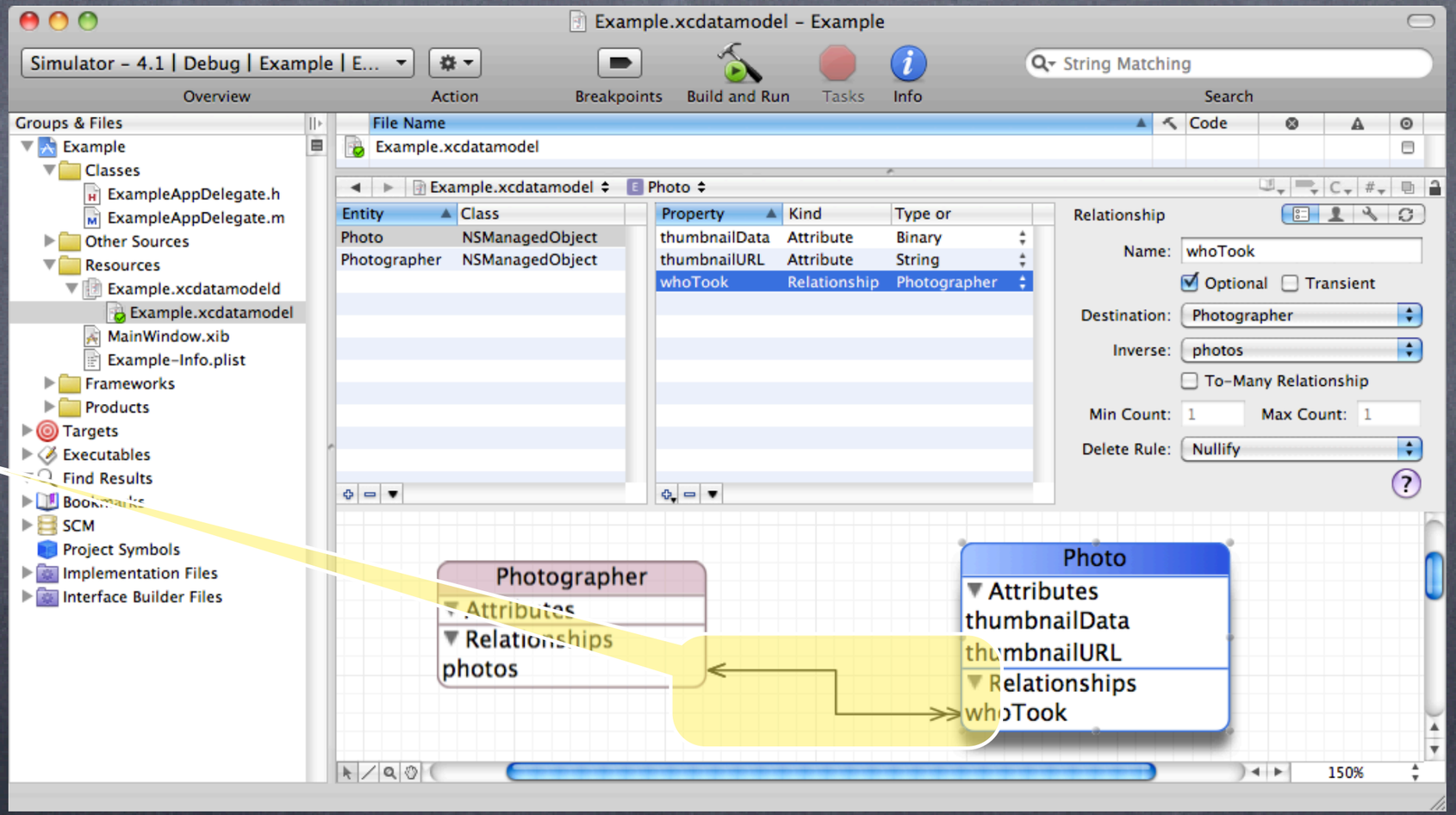
Core Data

- Relationships can go in both directions (inverse relationships)

  We've added the whoTook Relationship on Photo which points to a Photographer.

  whoTook is not a to-many relationship, so it will be an NSManagedObject property (not an NSSet).

The modeler now shows arrows in both directions for this Relationship.



Core Data

- There are lots of other things you can do
  But we are going to focus on creating Entities, attributes and relationships.
- So how do you access all of this stuff in your code?
- You need an `NSManagedObjectContext`
  It is the hub around which all Core Data activity turns.
- How do I get one?
  When you create your project, click the box that says "Use Core Data for storage."
  Your application delegate will then have some code in it to support Core Data.
  Most especially the application delegate `@property` called `managedObjectContext`.
  This is the `NSManagedObjectContext` you need to create/query objects in the database.

Core Data

- How to create a new object in the database

```
NSManagedObject *photo =
    [NSEntityDescription insertNewObjectForEntityForName:@“Photo”
                                  inManagedObjectContext:(NSManagedObjectContext *)ctxt];
```

Note that the class method above returns an `NSManagedObject`.
ALL objects from the database are either `NSManagedObject`s or subclasses thereof.
(More on that in a moment.)

- Passing `NSManagedObjectContext` through to where its needed

Usually the code above is not in your application delegate.
And you don't want to use your application delegate as a "global variable."
So you must pass the `NSManagedObjectContext` through APIs (to controllers, etc.).

Therefore, many (most?) Core Data-displaying view controllers' initializers look like this ...

```
- initInManagedContext:(NSManagedObjectContext *)context;
```

Core Data

- Now that I have created an object how do I get attribute values?
  You can access them using the following two methods …
  `- (id)valueForKey:(NSString *)key;`
  `- (void)setValue:(id)value forKey:(NSString *)key;`
- The `key` is an Attribute name in your data mapping
  For example, `@"thumbnailURL"`
- The `value` is whatever is stored (or to be stored) in the database
  It'll be `nil` if nothing has been set yet.
  Note that all values are objects (numbers and booleans are `NSNumber` objects).
  "To-many" mapped relationships are `NSSet` objects (non-"to-many" `values` are `NSManagedObjects`).
  Binary data values are `NSData` objects.
- Everything will be managed for you
  SQL statements will be generated automatically.
  Fetching will happen lazily as objects' properties are accessed/stored to.

Core Data

- Changes (writes) only happen in memory though, until you `save:`

  You should `save:` fairly often. Anytime a batch of changes occur.
  `save:` is an instance method in `NSManagedObjectContext` …

```
- (BOOL)save:(NSError **)errors;
```

  You can check first to see if there are any changes to objects in your `NSManagedObjectContext`.

```
- (BOOL)hasChanges;
```

  Here's the typical code to save changes after you update a value.
  If you pass `NULL` for the `(NSError **)`, it will just ignore all changes after the first error occurs.
  Otherwise, it will do everything it can and report all errors (via the `NSError`'s `userInfo`).

```
- (void)saveChangesToObjectsInMyMOC:(NSManagedObjectContext *)context {
    NSError *error = nil;
    if ([context hasChanges] && ![context save:&error]) {
        NSLog(@"Error! %@, %@", error, [error userInfo]);
        abort();  // generates a crash log for debugging purposes
    }
}
```

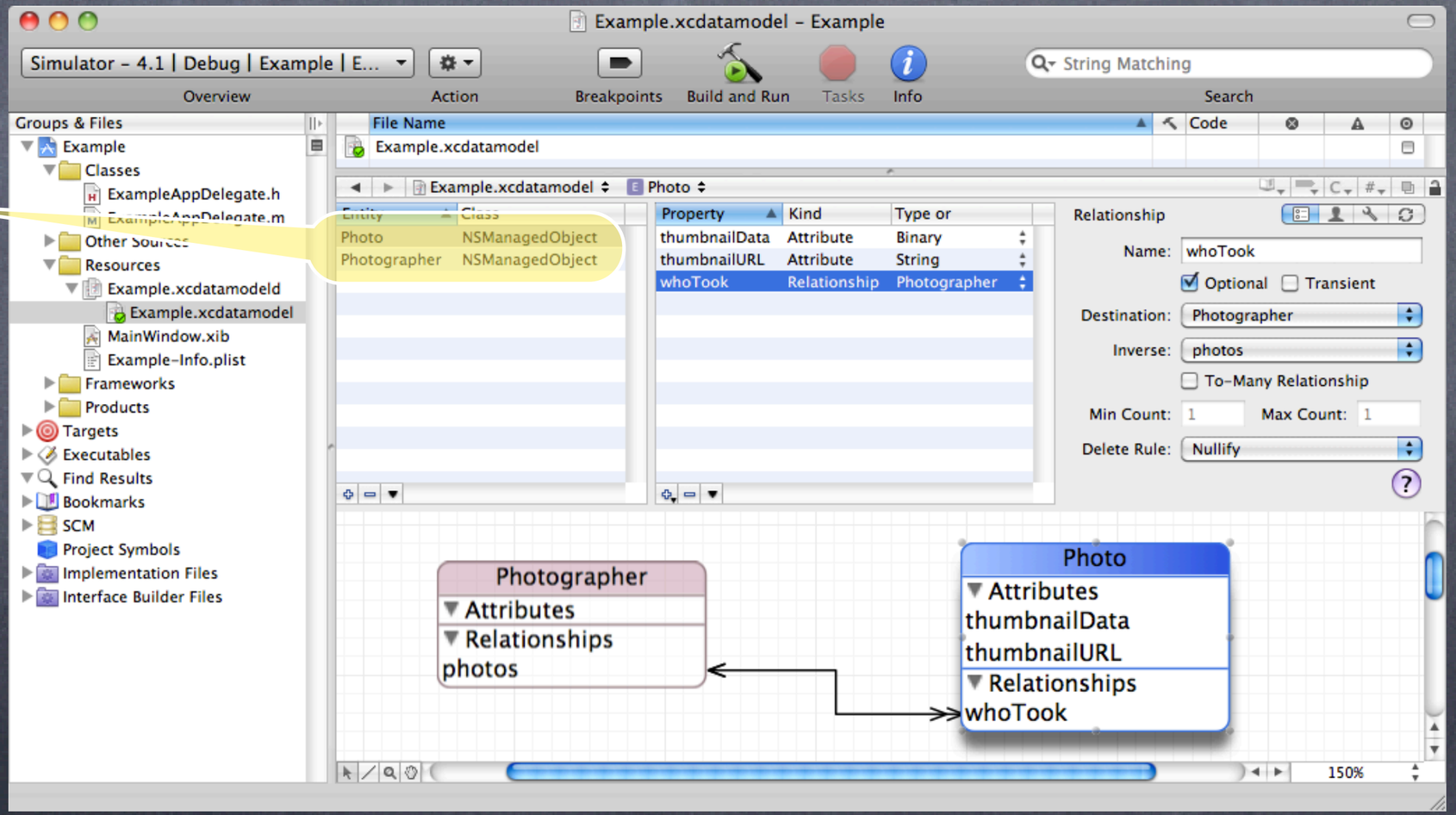
Core Data

- But calling `valueForKey:`/`setValueForKey:` is pretty messy
  - There's no type-checking.
  - And you have a lot of literal strings in your code (e.g. `@"thumbnailURL"`)
- What we really want is to set/get using `@property`s!
- No problem … we just create a subclass of `NSManagedObject`
  - The subclass will have `@property`s for each attribute in the database.
  - We name our subclass the same name as the Entity it matches (not strictly required, but do it).
  - And, as you might imagine, we can get Xcode to generate both the header file `@property` entries, and the corresponding implementation code (which is not `@synthesize`, so watch out!).
- There are two ways to get Xcode to generate this code
  - Create an entire new class from the description in the data mapping we created.
  - "Copy and paste" particular attributes as needed (e.g. new ones we've added lately).

Core Data

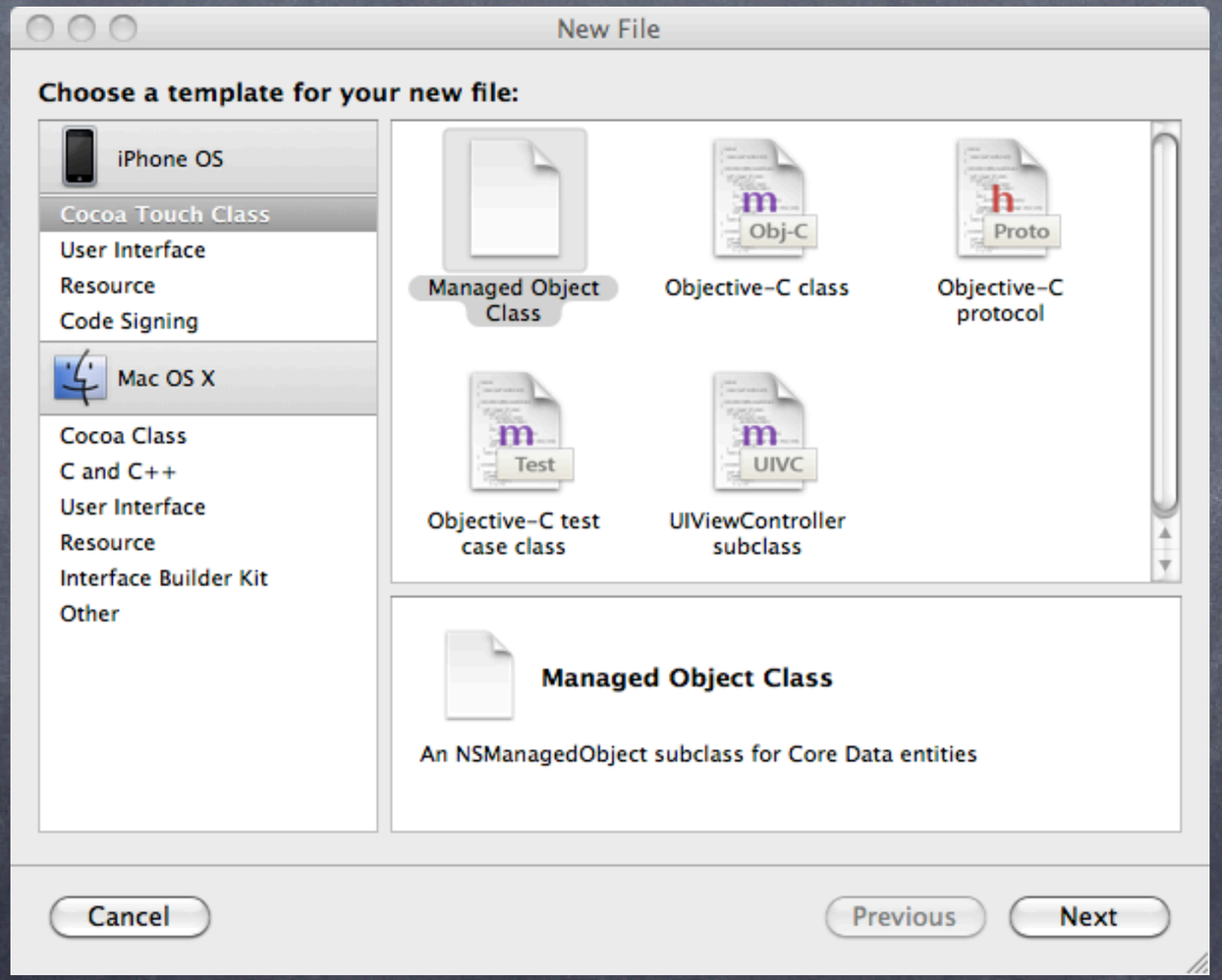
- Go back to Xcode and select one of your Entities
  If you don't, Xcode will not give you the option to create `NSManagedObject` subclasses.
  It doesn't matter which Entity you select because you're going to be given a choice in a moment.

Click on one of these.



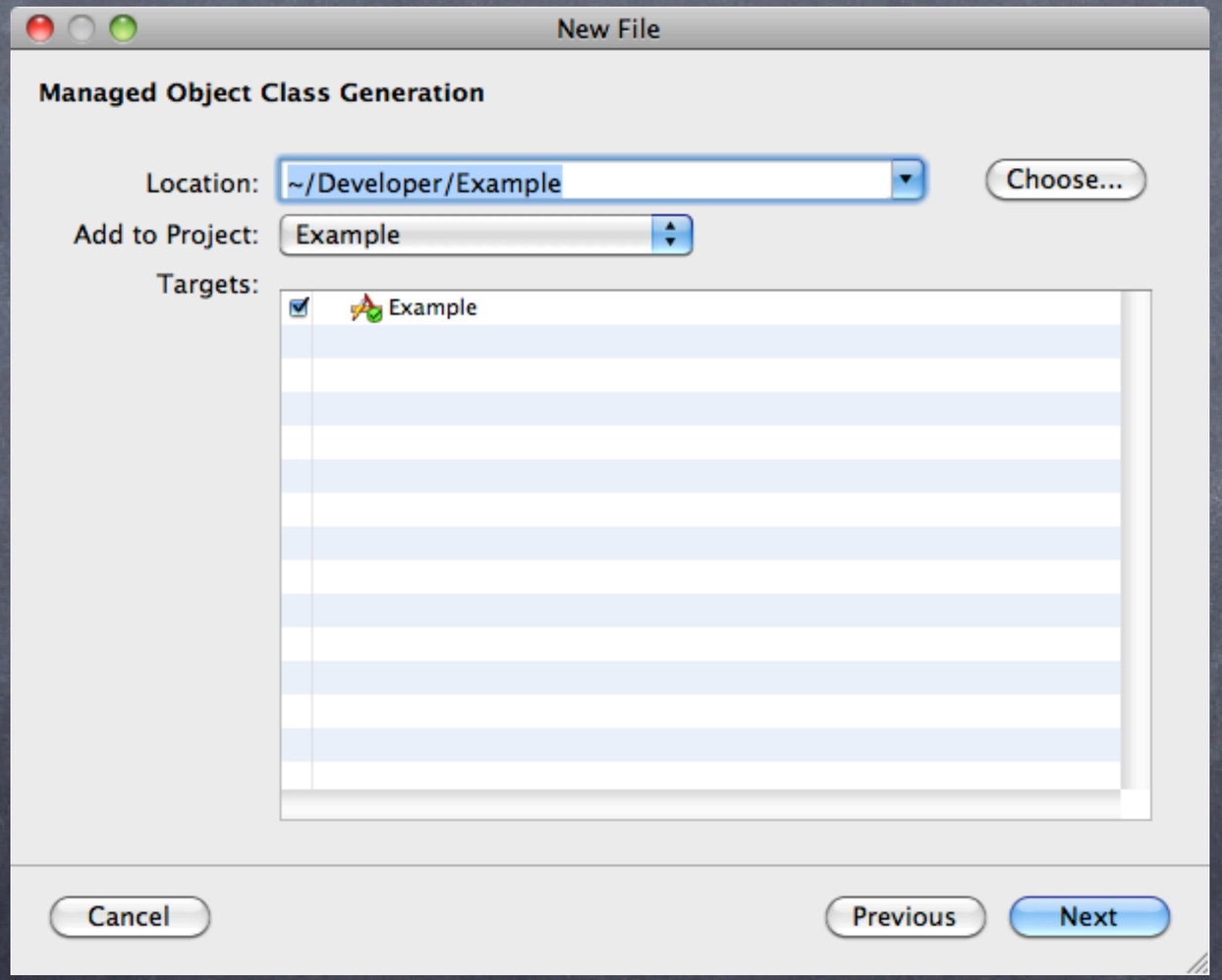
Core Data

- Choose New File … (because we're going to create a class)
  … and then choose Managed Object Class as the type of file to create.



Core Data

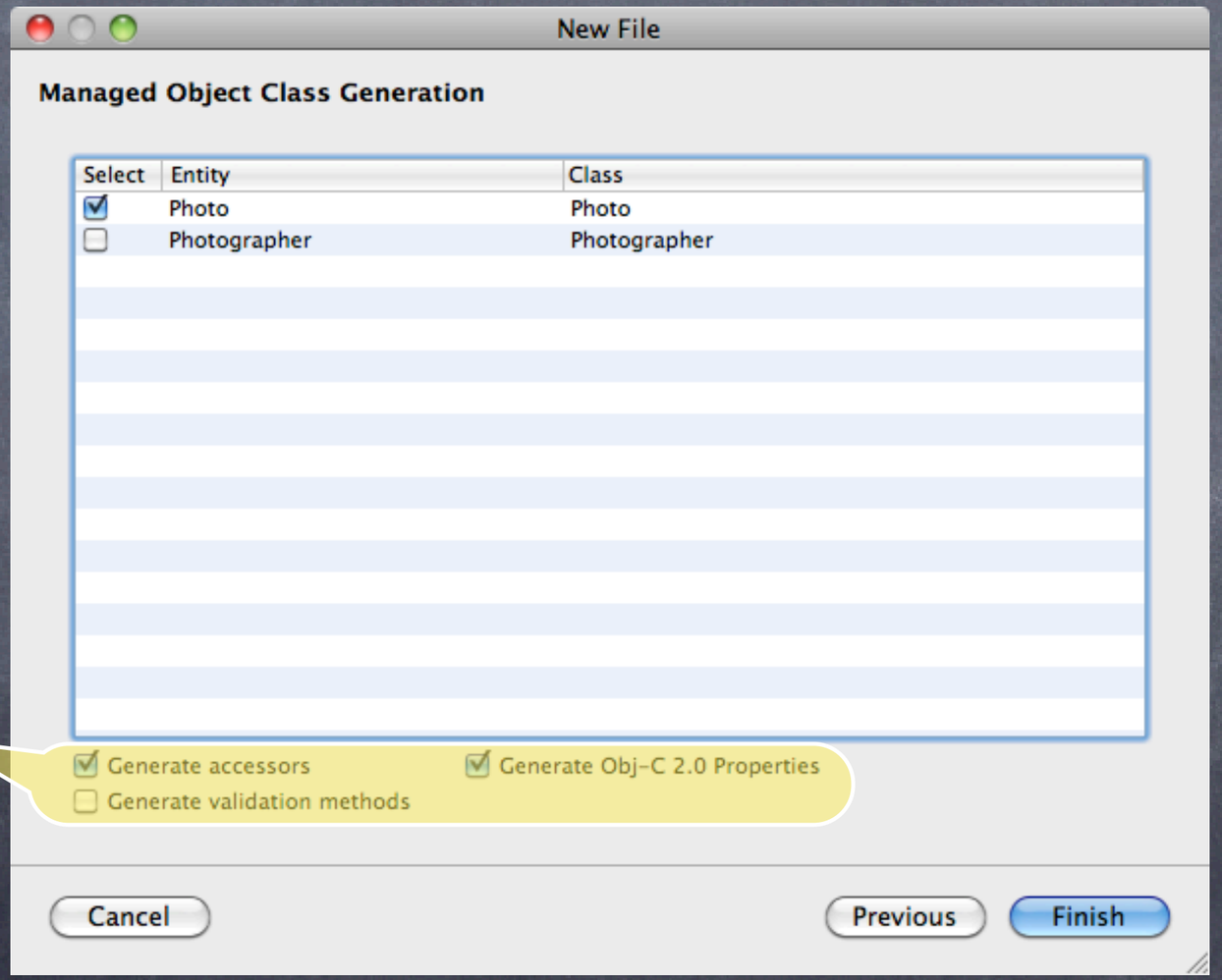
- It will then ask which project/directory to put the new class in
  Just click Next.



Core Data

- Finally it will ask which Entities you want to create classes for
  Select the ones for which you want custom subclasses of NSManagedObject to be created.

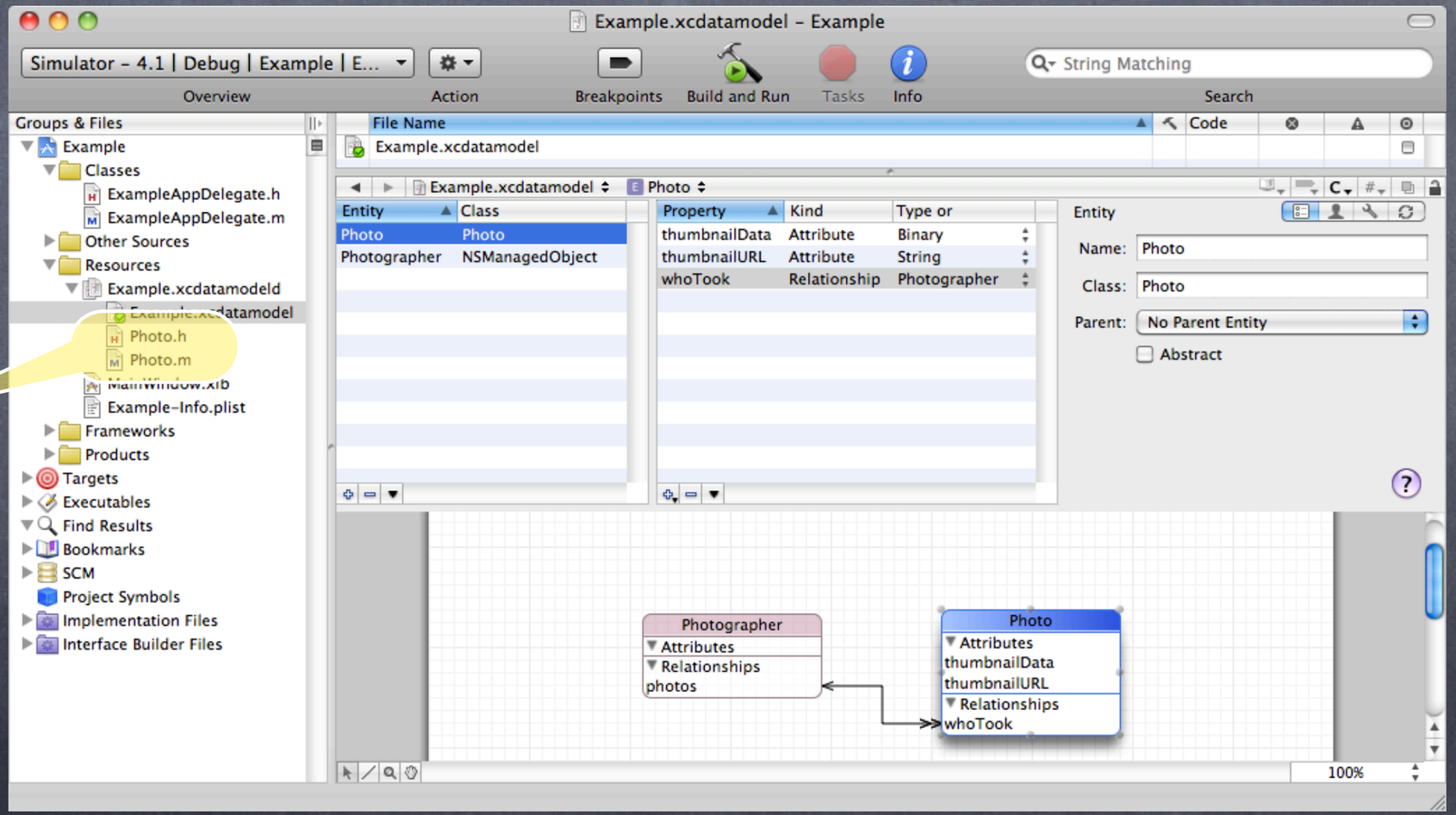
You'll want to leave these boxes checked as shown.



Core Data

- Files (`.h` and `.m`) for a new class will appear in `.xcdatamodeld`
  We'll take a look at the contents of them in a few slides.

Here are your new class's files.

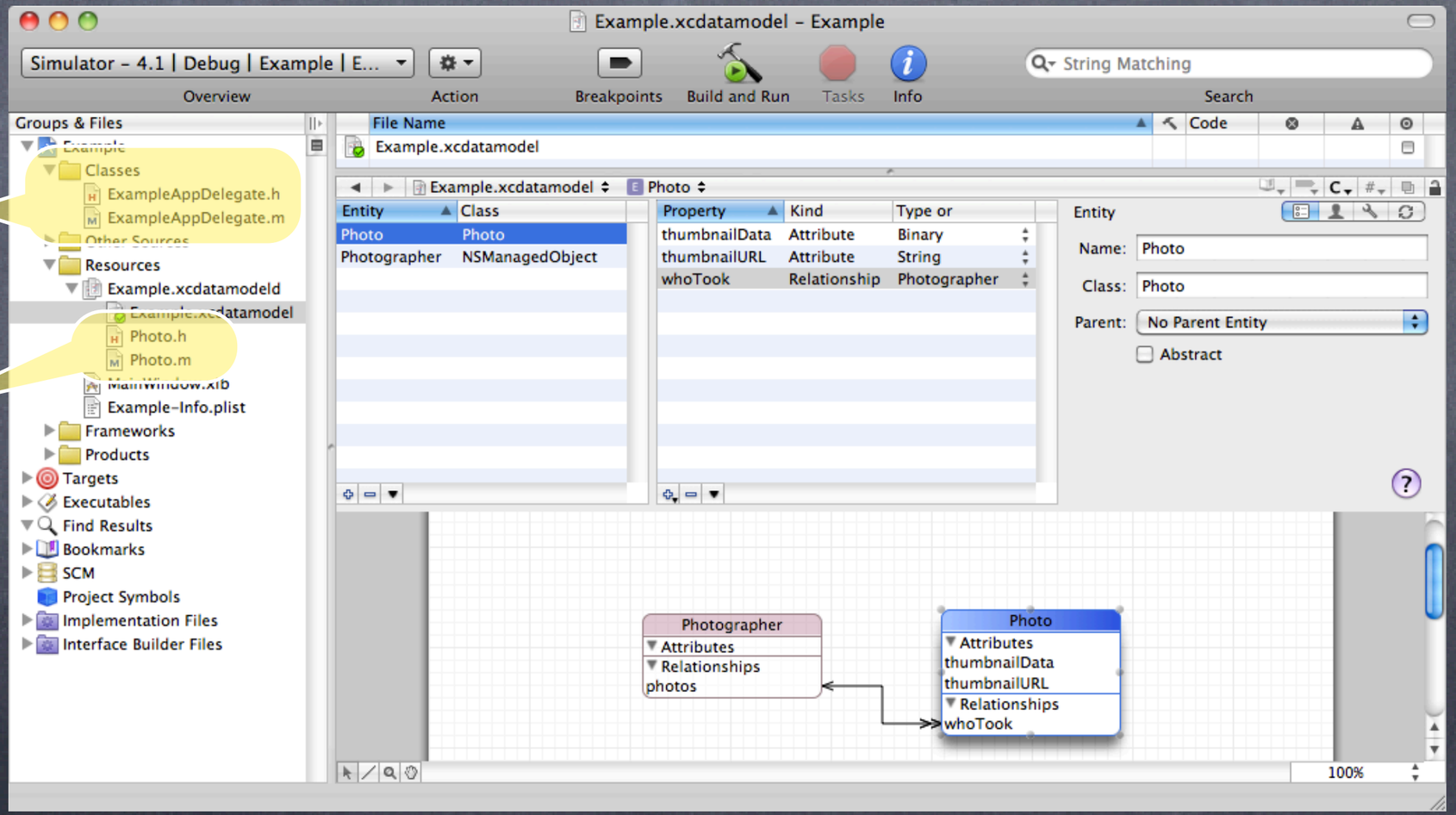


Core Data

- Files (`.h` and `.m`) for a new class will appear in `.xcdatamodeld`
  We'll take a look at the contents of them in a few slides.

You'll probably want to move them to Classes.

Here are your new class's files.

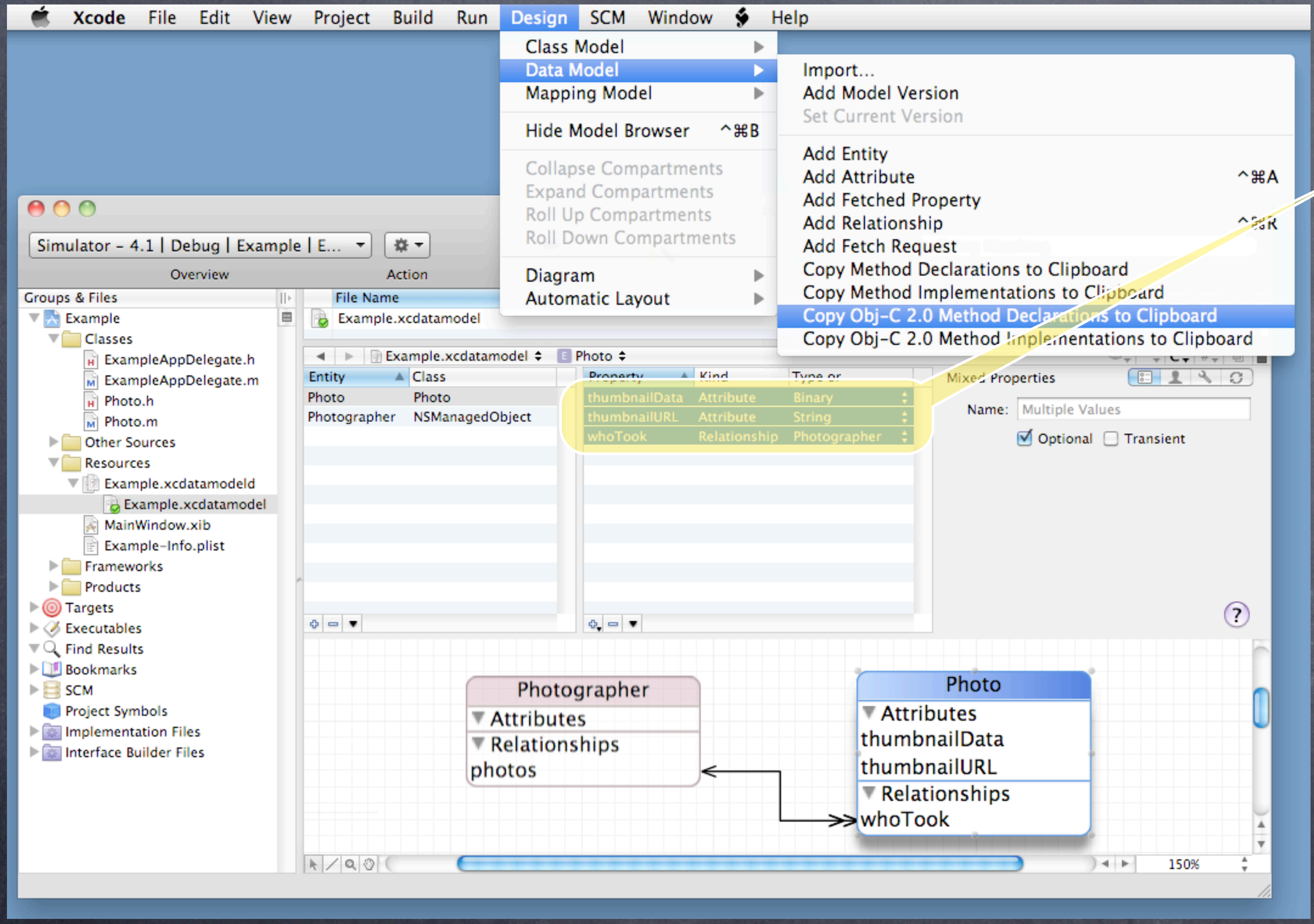


Core Data

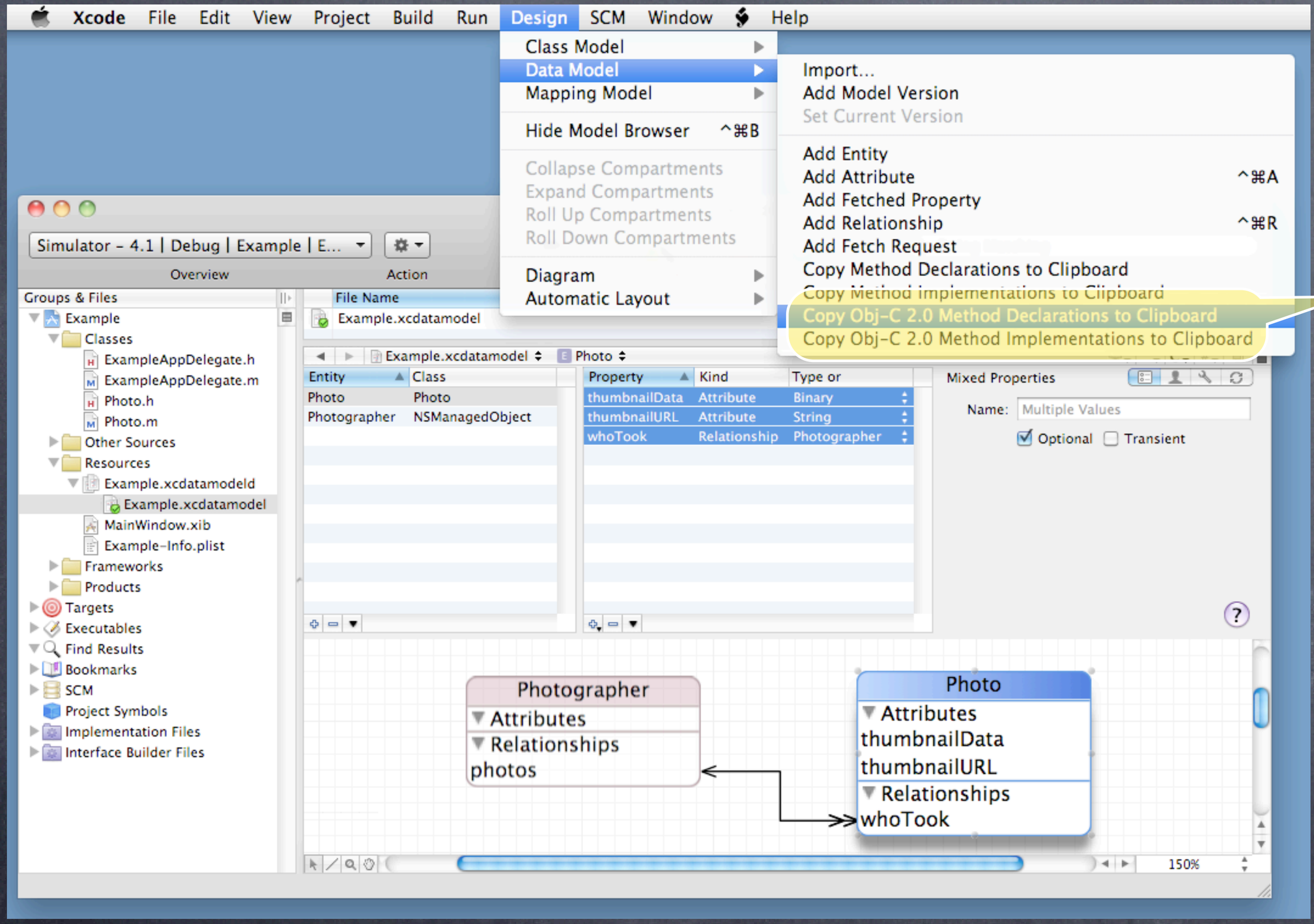
- Or you can create the code on a per-Attribute basis

  If you already have an `NSManagedObject` custom subclass
  (either because you created one manually or used the process on the previous few slides)
  you might add or change an attribute and need to generate code for that attribute only.

  The way you do this is to "copy and paste" the code from your data mapper to your subclass.
  You obviously need a special copy and paste menu item to do that.



Go to the data modeler and select any attributes you want Xcode to generate code for.



Go to the data modeler and select any attributes you want Xcode to generate code for.

Then copy either the declarations (header file) or implementations (implementation file) to the clipboard.

Then just go to your custom `NSManagedObject` subclass and paste them in (replacing old versions if necessary).

Coming Up

- Next Lecture
  Core Data and Table Views
  Big Demo
- Next Week
  Blocks and Multithreading
  Final Project Guidelines