Effective SOFTWARE DEVELOPMENT SERIES
Scott Meyers, Consulting Editor

Effective Objective-C 2.0

52 Specific Ways to Improve Your iOS and OS X Programs

Matt Galloway

Effective Objective-C 2.0

```
- (void)soundPlayerDidFinish:(EOCSoundPlayer*)player;
@end

@interface EOCSoundPlayer : NSObject
@property (nonatomic, weak) id <EOCSoundPlayerDelegate> delegate;
- (id)initWithURL:(NSURL*)url;
- (void)playSound;
@end

// EOCSoundPlayer.m
#import "EOCSoundPlayer.h"
#import <AudioToolbox/AudioToolbox.h>

void completion(SystemSoundID ssID, void *clientData) {
    EOCSoundPlayer *player =
        (__bridge EOCSoundPlayer*)clientData;
    if ([player.delegate
            respondsToSelector:@selector(soundPlayerDidFinish:)])
    {
        [player.delegate soundPlayerDidFinish:player];
    }
}

@implementation EOCSoundPlayer {
    SystemSoundID _systemSoundID;
}

- (id)initWithURL:(NSURL*)url {
    if ((self = [super init])) {
        AudioServicesCreateSystemSoundID((__bridge CFURLRef)url,
                                         &_systemSoundID);
    }
    return self;
}

- (void)dealloc {
    AudioServicesDisposeSystemSoundID(_systemSoundID);
}

- (void)playSound {
    AudioServicesAddSystemSoundCompletion(
        _systemSoundID,
        NULL,
        NULL,
        completion,
        (__bridge void*)self);
    AudioServicesPlaySystemSound(_systemSoundID);
}

@end
```

This looks completely innocuous, but looking at the symbol table for the object file created from this class, we find the following:

```
00000230 t -[EOCSoundPlayer .cxx_destruct]
0000014c t -[EOCSoundPlayer dealloc]
000001e0 t -[EOCSoundPlayer delegate]
0000009c t -[EOCSoundPlayer initWithURL:]
00000198 t -[EOCSoundPlayer playSound]
00000208 t -[EOCSoundPlayer setDelegate:]
00000b88 S _OBJC_CLASS_$_EOCSoundPlayer
00000bb8 S _OBJC_IVAR_$_EOCSoundPlayer._delegate
00000bb4 S _OBJC_IVAR_$_EOCSoundPlayer._systemSoundID
00000b9c S _OBJC_METACLASS_$_EOCSoundPlayer
00000000 T _completion
00000bf8 s l_OBJC_$_INSTANCE_METHODS_EOCSoundPlayer
00000c48 s l_OBJC_$_INSTANCE_VARIABLES_EOCSoundPlayer
00000c78 s l_OBJC_$_PROP_LIST_EOCSoundPlayer
00000c88 s l_OBJC_CLASS_RO_$_EOCSoundPlayer
00000bd0 s l_OBJC_METACLASS_RO_$_EOCSoundPlayer
```

Note the symbol lurking in the middle, called `_completion`. This is the completion function created to handle when the sound has finished playing. Even though it's defined in your implementation file and not declared in the header file, it still appears as a top-level symbol like this. Thus, if another function called `completion` is created somewhere, you'll end up with a duplicate-symbol error when linking, such as the following:

```
duplicate symbol _completion in:
    build/EOCSoundPlayer.o
    build/EOCAnotherClass.o
```

Worse still would be if you were shipping your code as a library for others to use in their own applications. If you exposed a symbol called `_completion` like this, anyone using your library would not be able to create a function called `completion`, which is a nasty thing for you to do.

So you should always prefix C functions like this as well. In the preceding example, you could call the completion handler

`EOCSoundPlayerCompletion`, for example. This also has the side effect that if the symbol ever appears in a backtrace, it's easy to determine what code the problem has come from.

You should be particularly careful of the duplicate-symbol problems when you use third-party libraries and ship your code as a library that others plug into their own applications. If the third-party libraries that you use are also going to be used by the application, it's easy for duplicate-symbol errors to arise. In that case, it's common to edit all the code of the libraries you use to prefix them with your own prefix. For example, if your library is called `EOCLibrary` and you pull in a library called `XYZLibrary`, you would go through and prefix all names in `XYZLibrary` with EOC. The application is then free to use `XYZLibrary` itself, without the chance of a naming collision, as shown in Figure 3.1.

Going through and changing all the names might seem like a rather tedious thing to do, but it's prudent if you want to avoid naming collisions. You may ask why it's necessary to do this at all and why the application can't simply not include `XYZLibrary` itself and use your implementation of it. It's possible to do that as well, but consider the scenario in which the application pulls in a third library, `ABCLibrary`, that has also decided to use `XYZLibrary`. In that case, if you and the author of `ABCLibrary` hadn't prefixed, the application would still get duplicate-symbol errors. Or if you use version X of `XYZLibrary` but the application requires features from version Y, it would want its own copy anyway. If you spend time using popular third-party libraries with iOS development, you will frequently see this kind of prefixing.

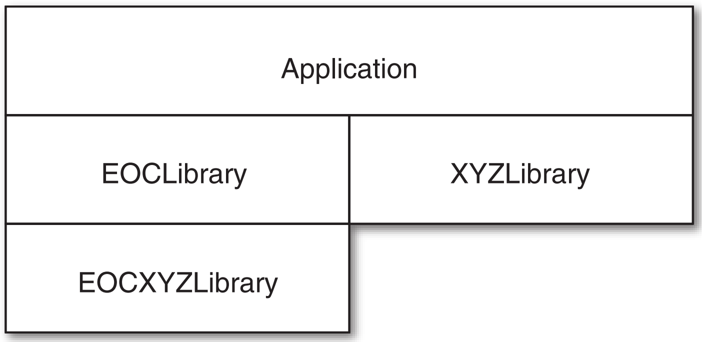

Figure 3.1 Avoiding naming clashes where a third-party library is included twice: once by the application itself and once by another library

Things to Remember

✦ Choose a class prefix that befits your company, application, or both. Then stick with that prefix throughout.

✦ If you use a third-party library as a dependency of your own library, consider prefixing its names with your own prefix.

Item 16: Have a Designated Initializer

All objects need to be initialized. When initializing an object, you sometimes don't need to give it any information, but often you do. This is usually the case if the object cannot perform its action without knowing this information. An example from UIKit, the iOS UI framework, is `UITableViewCell`, which requires being told its style and an identifier to group cells of different types, enabling efficient reuse of cell objects that are expensive to create. The term given to the initializer that gives the object the required amount of information to perform its task is the "designated initializer."

If there is more than one way to create an instance of a class, the class may have more than one initializer. This is perfectly fine, but there should still be just the one designated initializer through which all other initializers call. An example of this is `NSDate`, which has the following initializers:

```
- (id)init
- (id)initWithString:(NSString*)string
- (id)initWithTimeIntervalSinceNow:(NSTimeInterval)seconds
- (id)initWithTimeInterval:(NSTimeInterval)seconds
                 sinceDate:(NSDate*)refDate
- (id)initWithTimeIntervalSinceReferenceDate:
                              (NSTimeInterval)seconds
- (id)initWithTimeIntervalSince1970:(NSTimeInterval)seconds
```

The designated initializer in this case is `initWithTimeIntervalSinceReferenceDate:`, as explained in the documentation for the class. This means that all the other initializers call through to this initializer. Thus, the designated initializer is the only place where internal data is stored. If the underlying data store is changed for any reason, only one place needs to be changed.

For example, consider a class that represents a rectangle. The interface would look like this:

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

@interface EOCRectangle : NSObject
@property (nonatomic, assign, readonly) float width;
```

```
@property (nonatomic, assign, readonly) float height;
@end
```

Taking heed of Item 18, the properties are read-only. But that means that there's no way in which a Rectangle object can ever have its properties set externally. So you might introduce an initializer:

```
- (id)initWithWidth:(float)width
          andHeight:(float)height
{
    if ((self = [super init])) {
        _width = width;
        _height = height;
    }
    return self;
}
```

But what if someone decides to create a rectangle by calling [[EOCRectangle alloc] init]? Doing so is legitimate, since EOCRectangle's superclass, NSObject, implements a method called init, which simply sets all instance variables to 0 (or equivalent of 0 for that data type). If that method is invoked, the width and height will stay set at 0 from when the EOCRectangle instance was allocated (in the call to alloc) in which all instance variables are set to 0. Although this might be what you want, you will more likely want to either set the default values yourself or throw an exception to indicate that an instance must be intialized using your designated initializer only. In the case of EOCRectangle, this would mean overriding the init method, like so:

```
// Using default values
- (id)init {
    return [self initWithWidth:5.0f andHeight:10.0f];
}

// Throwing an exception
- (id)init {
    @throw [NSException
            exceptionWithName:NSInternalInconsistencyException
             reason:@"Must use initWithWidth:andHeight: instead."
           userInfo:nil];

}
```

Note how the version that sets default values calls straight through to the designated initializer. This version could have been implemented by directly setting the _width and _height instance variables.

However, if the storage backing the class had changed—for example, by using a structure to hold the width and height together—the logic of how to set the data would have to be changed in both initializers. This wouldn't be too bad in this simple example, but imagine a more complicated class with many more initializers and more complicated data. It would be very easy to forget to change one of the initializers and end up with inconsistencies.

Now imagine that you want to subclass EOCRectangle and create a class called EOCSquare. It's clear that this hierarchy makes sense, but what should the initializer be? Clearly, the width and height should be forced to be equal, since that's what a square is! So you might decide to create an initializer like so:

```
#import "EOCRectangle.h"

@interface EOCSquare : EOCRectangle
- (id)initWithDimension:(float)dimension;
@end

@implementation EOCSquare

- (id)initWithDimension:(float)dimension {
    return [super initWithWidth:dimension andHeight:dimension];
}

@end
```

This would become EOCSquare's designated initializer. Note how it calls its superclass's designated initializer. If you look back to the implementation of EOCRectangle, you'll see that it too calls its superclass's designated initializer. This chain of designated initializers is important to maintain. However, it's still possible for callers to initialize an EOCSquare object by using either the initWithWidth:andHeight: or the init method. You don't really want to allow this, since someone could end up creating a square whose width and height don't match. This illustrates an important point when subclassing. You should always override the designated initializer of your superclass if you have a designated initializer with a different name. In the case of EOCSquare, you could override EOCRectangle's designated initializer like so:

```
- (id)initWithWidth:(float)width andHeight:(float)height {
    float dimension = MAX(width, height);
    return [self initWithDimension:dimension];
}
```