Making Objective-C Classes look Beautiful

Asked 14 years ago Modified 9 years, 2 months ago Viewed 2k times

37

I wanted to ask you all for you opinions on code smells in Objective C, specifically Cocoa Touch. I'm working on a fairly complex game, and about to start the Great December Refactoring.

A good number of my classes, the models in particular, are full of methods that deal with internal business logic; I'll be hiding these in a private category, in my war against massive header files. Those private categories contain a large number of declarations, and this makes me feel uneasy... almost like Objective-C's out to make me feel guilty about all of these methods.

The more I refactor (a good thing!), the more I have to maintain all this duplication (not so good). It just feels wrong.

In a language like Ruby, the community puts a LOT of emphasis on very short, clear, beautiful methods. My question is, for Objective C (Cocoa Touch specifically), how long are your methods, how big are your controllers, and how many methods per class do you all find becomes typical in your projects? Are there any particularly nice, beautiful examples of Classes made up

of short methods in Objective C, or is that simply not an important part of the language's culture?

DISCLOSURE: I'm currently reading "The Little Schemer", which should explain my sadness, re: Objective C.

objective-c cocoa-touch cocoa-design-patterns

Share

Improve this question

Follow

edited Sep 27, 2015 at 2:27

Mogsdad

45.7k • 21 • 160 • 284

asked Dec 9, 2010 at 6:27

Sam Ritchie

11k • 4 • 43 • 53

Thanks! It became clear that this was a serious issue when I started thinking about how much longer it takes to answer obj-c questions than ruby questions. Grr... – Sam Ritchie Dec 9, 2010 at 6:33

You might be interested in this stack-exchange proposal. It's almost ready to begin beta, just needs a few more.
– greatwolf Jan 19, 2011 at 5:09

Victor, thanks so much, that looks great. I'm working on learning Clojure, now, and there simply aren't any other folks around to check work with. I've just committed!
– Sam Ritchie Jan 19, 2011 at 5:16

3 Answers

Sorted by: Highest score (default)

15

Beauty is subjective. For me, an Objective-C class is beautiful if it is readable (I know what it is supposed to do) and maintainable (I can see what parts are responsible for doing what). I also don't like to be thrown out of reading code by an unfamiliar idiom. Sort of like when you are reading a book and you read something that takes you out of the immersion and reminds you that you are reading.

You'll probably get lots of different, mutually exclusive advice, but here are my thoughts.

- Nothing wrong with private methods being in a private category. That's what it is there for. If you don't like the declarations clogging up the file either use code folding in the IDE, or have your extensions as a category in a different file.
- Group related methods together and mark them with `#pragma mark` statements
- Whatever code layout you use, consistency is important. Take a few minutes and write your own guidelines (here are mine) so if you forget what you are supposed to be doing you have a reference.
- The controller doesn't have to be the delegate and datasource you can always have other classes for these.

- Use descriptive names for methods and properties. Yes, you may document them, but you can't see documentation when Xcode applies code completion, where well named methods and properties pay off. Also, code comments get stale if they aren't updated while the code itself changes.
- Don't try and write clever code. You might think that it's better to chain a sequence of method calls on one line, but the compiler is better at optimising than you might think. It's okay to use temporary variables to hold values (mostly these are just pointers anyway, so relatively small) if it improves readability. **Write code for humans to read**.
- DRY applies to Objective-C as much as other languages. Don't be worried about refactoring code into more methods. There is nothing wrong with having lots of methods as long as they are useful.

Share Improve this answer Follow

edited Nov 30, 2012 at 14:44

answered Dec 9, 2010 at 11:47

Abizern

150k ● 41 ● 208 ● 260

9 I've given up `#pragma mark` in favour of `// MARK:` it has the same effect on the menu in Xcode but doesn't cause warnings in code that is portable between compilers. Also, `// MARK: -` is a real help (works with the pragma too). It puts a horizontal rule in the menu. – JeremyP Dec 9, 2010 at 14:20

Thanks for this, these were all great, and are coming in very handy during the Great Refactoring. – Sam Ritchie Dec 12, 2010 at 22:27

7

The very first thing I do even before implementing class or method is to ask: *"How would I want to use this from the outside?"*

I never ever, **never** begin by writing the internals of my classes and methods first. By starting of with an elegant public API the internals tend to become elegant for free, and if they don't then the ugliness is at least contained to a single method or class, and not allowed to pollute the rest of the code with it's smell.

There are many design patterns out there, two decades of coding have taught me that the only pattern that stand the test of time is: **KISS**. Keep It Simple Stupid.

Some general rules of thumb, for any language or environment:

- Follow your gut feeling over any advice you have read or heard!
- Bail out early!
    - If needed, verify inputs early and bail out fast! Less cleanup to do.
- Never add something to your code that you do not use.

    - An option for *"reverse"* might feel like something nice to have down the road.
    - In that case add it down the road! Do not waste time adding complexity you do not need.
- Method names should describe what is done, **never** how it is done.
    - Methods should be allowed to change their implementation without changing their name as long as the result is the same.
    - If you can not understand what a method does from it's name then change the name!
    - If the how part is complex enough, then use comments to describe your implementation.
- Do not fear the singletons!
    - If your app only have one data model, then it is a singleton!
    - Passing around a single variable all over the place is just pretending it is something else but a singleton and adding complexity as bonus.
- Plan for failures from the start.
    - Always use for `doFoo:error` instead of `doFoo:` from the start.
    - Create nice `NSError` instances with end user readable localized descriptions from the start.
    - It is a major pain to retrofit error handling/messages to a large existing app.

    - And there will always be errors if you have users and IO involved!
- Cocoa/Objective-C is **Object* Oriented, not **Class** Oriented as most of the popular kids out there that claim to be OOP.
    - Do not introduce a dumb value class with only properties, a class without methods performing actual work could just as well be a struct.
    - Let your objects be intelligent! Why add a whole new `FooParser` class if a `fooFromString:` method on `Foo` is all you need?
- In Cocoa what you *can do* is always more important than *what you are*.
    - Do not introduce a protocol if a target/action can do.
    - Do not verify that instances conforms to protocols, is a kind of class, that is up to the compiler.

Share Improve this answer

Follow

answered Dec 12, 2010 at 11:44

PeyloW

36.8k ● 12 ● 81 ● 99

Apple prefers proporties-only classes over structs (sorry, no reference). Also, introspection *can* be useful when dealing with `id`, subclasses or optional protocol methods.
– Scott Berrevoets May 22, 2013 at 23:06

@Scott - Jupp, this changes after the intro of ARC. ARC can not play well with object references in structs, but do with

classes. – PeyloW Jun 28, 2013 at 8:27

Completely agree with this, sums up my experience too :) – theLastNightTrain Jun 14, 2014 at 8:22

2

My 2 cents:

1. **Properties** are usually better than old-style getter+setter. Even if you use @dynamic properties - declare them with @property, this is way more informative and shorter.
2. I personally **don't simulate "private" methods** for classes. Yes, I can write a category somewhere in the .m(m) file, but since Obj-C has no pure way to declare a private method - why should I invent one? Anyway, even if you really need something like that - declare a separate "MyClassPrivate.h" with a category and include it in the .m(m) files to avoid duplicating the declarations.
3. **Binding**. Binding for most Controller <-> UI relations, use transformers, formatters, just don't write methods to read/write controls values manually. It makes code look like something from MFC era.
4. **C++**, a lot of code look much better and shorter when written in C++. Since compiler understands C++ classes it's a good point for refactoring, especially when working will a low-level code.
5. I usually split big controllers. Something **more than 500 lines of code** is a good candidate for refactoring

for me. For instance, I have a document window controller, since some version of the app it extends with image importing/exporting options. Controller grows up to 1.000 lines where 1/2 is the "image stuff". That's a "trigger" for me to make an ImageStuffController, instantiate it in the NIB and put all image-relative code in there.

All above make it easier for me to maintain my code. For a huge projects, where splitting the controllers and classes to keep 'em small results big number of files, I usually try to extract some code into a framework. For example, if a big part of the app is communicating with external web-services, there is usually a straight way to extract a MyWebServices.framework from the main app.

Share Improve this answer

Follow

answered Dec 9, 2010 at 11:24

Gobra

4,261 ● 2 ● 16 ● 20

12 #4 is a terrible, terrible idea. First time I've heard anyone say to translate to C++ to make code look better. – Jon Shier Dec 9, 2010 at 20:07

1 jshier, C++ allows to define a + operator for NSPoint, for example. For me, p1 + p2 looks a way better than NSOffsetPoint(p1, p2). C++ is much cleaner when working with data structures like maps and list (unless result is directly used in UI - that's where you should go with NS-classes). Next - math tasks. C++ library for matricies and vectors is usually muuuch easier to use than NSAffineTransform and other. NS-classes are yet again good when working with NSGraphicsContext, but some back-end math should definitely be processed with C++ to

simplify the code. IMHO, of course – Gobra Dec 10, 2010 at 10:28