Refactoring iOS Projects

Adrian Kosmaczewski

2016-07-17

Presentation given in Dnipropetrovsk, Ukraine, on July 16th, 2016. In this session we are going to learn simple yet effective techniques to refactor large iOS codebases in order to make them more testable, to adapt them to be eventually rewritten in Swift, and to make them as "future proof" as possible.

Hi everyone, thanks for choosing to attend my session and thanks to the organisers of the conference for the opportunity to talk about this subject.

So, refactoring iOS projects. Refactoring is available in Xcode on the `Edit` menu, and also when you right-click on the source code in the editor.

Thank you so much for your attention.

Oh, sorry.

Of course, we all know that AppCode is much better for refactoring. Actually it is the number one feature that every AppCode user ever mentions to me when they evangelize AppCode. It is so good that there is even a page in the JetBrains website that talks about it!

In that page you can watch the difference between both IDEs.

Seriously?

Is that all? Is refactoring just a menu entry in an IDE? Is it just a marketing thing? Do we choose IDEs because of refactoring? Do we choose programming languages because of the IDEs? Do we choose the platforms we love because of the programming languages?

Well, yes and no. There is a lot of both, actually. To be a developer is also to wear the t-shirt of your favorite operating system, programming language and IDE of choice.

For example in my case it is macOS, C++ and Vim. But each one of you will have their own preferences.

Big Topic

Refactoring is a big topic indeed. While I was searching material for this presentation, I came across a rather large number of books with the title "Refactoring" in it, and clearly you can find a refactoring book for pretty much any programming language these days.

But in my view the most important of all of these is the original book by Martin Fowler, originally published in 1999 and that I am sure you have a copy in your own bookshelf, even if you never took the time to actually read it.

Refactoring became an industry, actually, and during the first few years of the 2000s we saw a family of "refactoring" books, first a few… then a lot…

So I decided to go back to the roots of this book, originally written with Java code examples. See, Java was the Swift of 1999, in every sense of the term, because even IBM was using it everywhere just like they are using Swift now.

But at the same time it has not, and we still need to refactor.

What is Refactoring?

The official definition of Refactoring is in page 53 of the original book:

> Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.

From the noun we have also a verb, "to refactor."

Why Refactoring?

There are several reasons to refactor software:

- To improve the design of software
- To make it easier to understand
- To make it easier to find and fix bugs
- To program faster

When to refactor?

Do not set time aside for refactoring: just do it.

- Refactor when you add features
- Refactor when fixing bugs
- Refactor during code reviews

But do not refactor when you agree that you should rewrite from scratch. Sometimes code is not even worth of a refactor. Of course, remember Netscape before doing a full rewrite of your software.

Bad Smells

The original Refactoring book has a long list of "typical" bad smells, mostly discovered and based on Java or C++ code, that can be corrected by refactoring techniques. Here is the full list of these refactoring, as catalogued by Fowler:

- Duplicated code

- Long methods
- Large class
- Long parameter list
- Divergent change or Shotgun surgery
- Feature envy
- Data clumps
- Primitive obsession
- Switch statements
- Parallel inheritance hierarchies
- Lazy class
- Speculative generality
- Temporary field
- Message chains
- Middle Man
- Inappropriate intimacy
- Alternative classes with different interfaces
- Incomplete library class
- Data class
- Refused bequest
- Comments

Requirements for Refactoring

Testing. The most important thing to know about refactoring is that there is no point on doing any refactoring if you have not taken the time to write any unit or functional tests around your code.

So if your projects do not have extensive unit testing, I suggest that you stop everything that you are doing, and that you add them to your project before doing any refactoring. It is an order. And if your manager tells you that there is no time for that, that the customer does not pay for that, that the project is small enough, that this is just a prototype, and prevents you from writing tests, well then change jobs, because you are in the wrong place.

Unit testing and refactoring are the basic pillars of any valid QA strategy, and I know by experience that non-technical project managers very often fail to understand that developers must do these tasks, and that they are as important as writing the code itself.

Specific iOS Smells

The original Refactoring book uses Java for all of its code examples. Java is quite a rigid language, much more than Swift anyway. In particular it has only recently included functional features (lambdas exist only since Java 8) so at the time the book was originally written, it was quite a pure object-oriented language only.

In my professional life, I often had to review code of other iOS developers, and in some cases I had to help some of these developers to finish their projects and clean up their code. I have done this regularly since at least 2009 and I can positively say that I have seen a bit of everything. This talk will be a short summary of some common problems I have seen in iOS projects, and some simple measures to solve those problems. The idea is to make iOS projects future proof at any given time.

In the code samples of this presentation I will use Swift 3.0 on Xcode 8.

Some typical smells in iOS projects are the following, grouped as follows:

- Programming language smells.
- Class design smells.
- Project management smells.

1. Programming Language and Cocoa Smells

- SwiftyLeaks
- Hungarian Notation
- Main Thread Fatigue
- Objective-C Nostalgia
- Homemade Cache
- Tagged View
- Illicit Association
- Long Method Names

1.1 SwiftyLeaks This is a very important point, so I am going to go through it briefly just to get the idea straight for everyone.

Programming languages are nothing else than the description of a series of mutations in a memory space, executed by a CPU. These mutations are carried by operating system processes. Processes in iOS, just as in most modern operating systems (yes, this is true from Windows to Unixes to Android to…) divide the memory allocated in several different segments:

- The TEXT segment contains the binary of the application, running in memory.
- The DATA segment contains all the static variables you have declared in your code.
- Each thread of execution has its own STACK; this segment contains the local variables and is divided in "frames"; every time you enter a function or call a method, the CPU "push" a frame on the stack, where all the local variables are created. After the `return` statement, the frame is "popped" and all of its contents are lost. This is the reason why in C++ one does not return a pointer or a reference to a local variable; they will not exist after the method exits. In Objective-C, not only values exist in the stack,

but also `struct` and blocks. In C++ you can create objects on the stack just as you would create any variable.
- Finally, the HEAP segment, shared among all threads, contains all the memory that is allocated using `malloc()`, `calloc()` and `ralloc()`. All Objective-C objects created using `[[ClassName alloc] init]` exist in the heap exclusively. In C++ you can create objects on the heap using the `new` statement. You reference objects in memory on the heap using "pointers" (and "references" in C++, which are a special type of pointer that simulates value semantics) and of course, if you lose the pointer, then the memory on the heap can never be reclaimed, and we have this situation known as a *memory leak.*

In Objective-C, blocks are the only type of object created on the stack. They are automatically and transparently copied to the heap as soon as they are returned from a function, and then they are passed around as references.

In Swift the situation is less similar to Objective-C and more similar to Java or C#. Value types like primitives and `struct` could be allocated on the stack, and reference types could be allocated on the heap, but they could end up on another segment for performance purposes. This is something that the runtime manages completely for you. The Java JVM and the .NET CLR do these kind of optimisations in the background as well, and usually with excellent results that you get for free. Read more about this.

The important thing to know here has to do with language interoperability. If you still need to use Objective-C code in your future applications, remember that it allocates objects on the heap with the sole exception of blocks. In the case of Swift, `enum` and `struct` types are allocated on the stack.

In Swift, there are only two big reasons to consider if you need to choose between `class` and `struct`; choose classes if:

- You need inheritance
- You need reference semantics to share state
- You need to use Swift types in Objective-C.

In all the other cases, use structs. Just for the sake of code interoperability, please find below the features of Swift not available in Objective-C:

- Tuples
- Enumerations
- Structures
- Top-level functions
- Global variables
- Typealiases
- Swift-style variadics
- Nested types
- Curried functions

This list originally included Generics too, but they have been added to Objective-C a bit later after the appearance of Swift.

1.2 Hungarian Notation This is a very common problem, and the source of endless jokes. You can tell that a C# developer has been writing Objective-C because method names start in uppercase, or that a Java developer was writing Swift because there is an abstract factory for every single class in the system.

Please pay attention to the naming conventions of the language you are using, and adopt them. This is useful for many reasons:

- It helps in the long term for the maintenance of the software.
- It helps new team members read the existing code.
- If your system ever ends in the open source domain, other people out there will have less trouble understanding your code.

> Programs must be written for people to read, and only incidentally for machines to execute.

Harold Abelson, "Structure and Interpretation of Computer Programs"

So here go some useful links:

- Cocoa naming guidelines
- Objective-C naming guidelines
- Swift API design guidelines

Particularly for Swift 3, the main guideline to remember is to **Make uses of your APIs read grammatically:**

```
friends.remove(ted)
mainView.addChild(button, at: origin)
truck.removeBoxes(withLabel: "WWDC 2016")
```

Also name methods based on their side effects: *use verbs to describe the side effects*:

```
friends.reverse()
viewController.present(animated: true)
```

And use nouns to describe the result or what is being returned:

```
button.backgroundTitle(for:.disabled)
friends.suffix(3)
```

Also, pay attention to the "mutating / non-mutating pairs" of methods:

```
x.reverse() // mutating
let y = x.reversed() // non-mutating

dir.appendPathComponent(".txt") // mutating
let newDir = dir.appendingPathComponent(".txt") // non-mutating
```

Starting in iOS 10, every API will have two names:

1. One for Objective-C & Swift 2
2. Another for Swift 3

Swift 3 APIs can be annotated with the `@objc()` attribute, specifying a method name to be used with Objective-C.

Inversely, when exporting Objective-C APIs to Swift 3, you can now use the `NS_SWIFT_NAME()` macro to specify the names that you want to use in the newest version of the language.

Please watch session 403 of WWDC 2016 for more information about this subject.

1.3 Main Thread Fatigue This is a common mistake but relatively easy to solve these days, primarily thanks to GCD.

Four things to keep in mind:

- Separate Core Data MOC for background queues and threads
- Number crunching, filters, etc, use GCD
- Network operations, use the `NSURLSession` class family.
- Main thread === UI thread. Nothing else. Everything that cannot be seen should happen on a background GCD queue.

Pay attention to the new syntax of GCD APIs in Swift 3!

1.4 Objective-C Nostalgia You should seriously start thinking about migrating your Objective-C code to Swift in the near future. Swift 3 is going to be ABI compabible with Swift 4 (that is at least the explicit wish of the Swift team at Apple) so you should start adapting your Objective-C code so that it can be used by Swift classes.

- Use the `nonnull`, `nullable` and similar declarations
- Use generics for your containers:

```
NSArray<NSString *> *arrayOfString = @[@"one", @"two"];
```

Beware of the following caveats during the migration process:

- Objective-C cannot subclass a Swift class, unless it is marked with the `@objc` decoration.
- Tuples, Swift enums and structs are not accessible from Objective-C.

Objective-C will remain in my opinion a very useful language to wrap old C APIs and libraries, in order to make them accessible from Swift.

A special comment about the use of `#define` in Objective-C. This is a pet peeve of mine. Instead of

```
#define RADIUS 45.5
#define NAME_DICT_KEY @"name"
```

Do this:

```
static CGFloat RADIUS = 45.5;
static NSString * const NAME_DICT_KEY = @"name";
```

The above declarations only work when the symbols are used in the same compilation unit (read, `.m` file) where they are defined; if you need constants to reuse, do this:

In the header file:

```
NS_EXTERN NSString * const NAME_DICT_KEY;
```

and in the implementation file:

```
NSString * const NAME_DICT_KEY = @"name";
```

In Swift, needless to say, you can use nested enums inside of your structs and classes to define your own constants.

1.5 Homemade Cache Over the years I have seen a lot of applications that feature some kind of homemade cache object, to store images downloaded from the network or other artifacts. Creating these objects is really complex and requires lots of attention and testing… and you do not need to do that.

Just use `NSCache`. As easy as that. The problem with `NSCache` is that it has a relatively complex API documentation, so it is sometimes complicated to know which options to pass to the object initializer.

Fear no more, the code for a nice and simple instance of `NSCache` is here.

```
let config = URLSessionConfiguration.default()
config.requestCachePolicy = .returnCacheDataElseLoad
let memoryCapacity = 10 * 1024 * 1024;
let diskCapacity = 20 * 1024 * 1024;
let cache = URLCache(memoryCapacity: memoryCapacity,
                        diskCapacity: diskCapacity,
                        diskPath: nil)
URLCache.setShared(cache)
```

1.6 Tagged View There used to be a time when this was acceptable code:

```
override func tableView(_ tableView: UITableView,
                        cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: "Cell",
                                             for: indexPath)

    let object = objects[indexPath.row] as! NSDate
    cell.textLabel!.text = object.description

    let switchControl = cell.viewWithTag(1) as! UISwitch
    switchControl.setOn(false, animated: false)

    return cell
}
```

This is no longer acceptable. Please do not use tags like this. It leads to absolutely unmanageable code and it can lead to weird bugs. Particularly if your code uses magic strings and numbers all over the place.

And please do not use Swift `enum` to replace those tags. Just do not use tags.

```
override func tableView(_ tableView: UITableView,
                        cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let cell = tableView.dequeueReusableCell(withIdentifier: CellIdentifier,
                                                                    for: indexPath) as! CustomVi

    let object = objects[indexPath.row] as! NSDate

    cell.titleLabel.text = object.description
    cell.switchControl.setOn(true, animated: false)

    return cell
}
```

And the definition of the `CustomViewCell` could not be simpler:

```
class CustomViewCell: UITableViewCell {
    @IBOutlet weak var switchControl: UISwitch!
    @IBOutlet weak var titleLabel: UILabel!
}
```

And you should of course make the connections in Interface Builder, as required.

1.7 Illicit Association Associated objects are a great thing.

```
import Foundation

class SomeObject : NSObject { }

extension SomeObject {
    private struct AssociatedKeys {
        static var AssociatedValue = "AssociatedValue"
    }

    var associatedString: String? {
        get {
            return objc_getAssociatedObject(self,
                                            &AssociatedKeys.AssociatedValue) as? String
        }

        set {
            objc_setAssociatedObject(
                self,
                &AssociatedKeys.AssociatedValue,
                newValue as NSString?,
                .OBJC_ASSOCIATION_COPY_NONATOMIC
            )
        }
    }
}

var obj = SomeObject()
obj.associatedString = "Some other string"
print(obj.associatedString)
obj.associatedString = nil
print(obj.associatedString)
```

1.8 Long Method Names Old Objective-C APIs, when translated verbatim into Swift, will inevitably look verbose. Think about renaming them.

2. Class Design Smells

- Massive View Controller (MVC)
- Massive App Delegate (MAD)
- Forgotten Memory Warnings
- Long Switch Statement
- Excessive Curiosity
- Selfish Navigation
- Offline Confusion

2.1 Massive View Controller (MVC) This is by far the biggest problem I have seen in large iOS projects, one that I have been the culprit of causing several times, and I am pretty sure you have had this problem too in your own code at least once.

`UIViewController` is a fundamental piece in the UIKit framework. It can do everything, and sometimes it ends up doing too much. This situation must be prevented at all costs, because it makes very difficult to maintain large iOS applications – and all applications end up being large at some point.

Several strategies to break up controllers:

- Categories: break up your controllers in separate categories with distinct responsibilities.
- Helper objects: break up your controllers and create private, helper objects that take care of different functionalities.
- State pattern: if your controllers must act differently depending on their state, do not write `if` statements for that! Never, never, never. Use the state pattern. This requires you creating a separate class for each different state of your controller, and then perform the changes inside of those state subclasses. The GameplayKit framework contains since iOS 8 a very handy `GKStateMachine` class, with a matching `GKState` class, that you can subclass, and it offers a wonderful infrastructure for a very simple and straightforward implementation of the state pattern.

For those looking for a simpler approach, you can use a very simple Swift enum that takes a block as its associated values, and then you just execute the associated blocks. This provides a simple approach that allows you to organise your code.

The objective of breaking controllers down must be that the files defining them are never bigger than 400 lines of code. This is already a sizeable amount of code for a class, but it is manageable.

Check out this blog post for more alternative architectures.

2.2 Massive App Delegate (MAD) This problem, somehow related to the previous one, is the result of several factors:

1. Using Apple templates, particularly when selecting the "Core Data" option in Xcode
2. Not knowing what the App Delegate is there for

The application delegate is an ear. It listens for events from the application object, many of which come directly from the operating system itself. The idea of the app delegate is that it exists only so that you can have a centralized location for all the events that come from the operating system during the lifetime of the application. **And nothing else.**

This way your code knows when your application has started, with what parameters, when it is going to be sent to the background, when it is going to be killed, when there is an incoming memory warning, and things like that.

All code that does not explicitly pay attention to these events should not exist in the App Delegate. Please pay attention to this fact, and create separate objects to take care of different concerns.

This is a basic design issue, one that can be solved easily, yet I keep seeing it everywhere. My personal solution would be to avoid the default templates of Xcode, unless you are building a proof-of-concept or a prototype. Big projects should start as minimalist as possible, simply selecting the "Single View" template.

2.3 Forgotten Memory Warnings There are three types of memory warnings:

1. The App Delegate features the `applicationDidReceiveMemoryWarning(_ application:)` method.
2. Every `UIViewController` subclass should `override func didReceiveMemoryWarning()`
3. Finally, any object can listen to the `UIApplicationDidReceiveMemoryWarning` notification and respond accordingly.

But the sad truth is that most apps just choose to ignore *all* of them. Come on, we have all been guilty about this. And of course we can argue that the latest devices have more and more RAM. Of course the latest iPad Pro boasts an incredible 4 GB of RAM, so we might think that we just do not need to care about this anymore.

False.

Listen to all of these notifications. Implement all the methods. Clean any data that can be re-calculated later on, save files, bring down any complex data structures you will not need in the short term and that can be recreated or reloaded later. iOS still sends memory warnings, even in devices with lots of RAM, because memory management. And one cannot know exactly when or why this is going to happen, but be sure that this is going to happen sooner than later.

Lazy-loading is your friend. You can implement a public calculated property wrapping a private field, which has the advantage over `lazy var` statements in that their contents can be cleaned up at any moment.

2.4 Long Switch Statement This is a smell that was identified by Fowler in his book, but it is so prevalent that I would like to spend some time discussing some solution for this smell.

Long switch statements are drag. You all know that. However, for one reason or another, we keep doing them. Of course we are very happy now that Swift implements a safe `switch` statement, one in which one has to actually write the `fallthrough` keyword to jump from case to case.

On the other hand, Swift brings so many niceties to `switch` statements, such as pattern matching, that it is very tempting to use them and to use the often.

But they are drag for several reasons:

- They increase the number of paths in the code, making it harder to test.
- They can decrease the readability of the code when they span several screens.
- They can contribute to the MVC smell (Massive View Controller.)

Three strategies to solve this problem:

1. Extract methods for each switch statement first.
2. Move those methods to separate objects.
3. Remove the switch statement altogether with polymorphism
4. Use the state, decorator or strategy patterns for more complex situations.

Using these strategies you will ensure that your code is maintainable and testable.

2.5 Excessive Curiosity If you see any of the following in a code, this is a rather strong smell:

```
// Only if you need compatibility with iPhone OS 3.2... really?
if UI_USER_INTERFACE_IDIOM() == UIUserInterfaceIdiom.pad {
    // ... iPad-only code
}


let idiom = UIDevice.current().userInterfaceIdiom
if idiom == UIUserInterfaceIdiom.phone {
    // ... iPhone-only code
}
```

To begin with, you should encapsulate iPad-only or iPhone-only code in universal apps in a separate class, and test it accordingly. This includes controllers, helpers, even models that might run in another context. If the amount of code is too much, you might want to separate it in its own framework, but again, pay attention to testability and simplicity.

If you still want to do version checking, since iOS 8 you can use `NSProcessInfo` like this:

```
// Since iOS 8:
let version = OperatingSystemVersion(majorVersion: 10, minorVersion: 0, patchVersion: 0)
if ProcessInfo().isOperatingSystemAtLeast(version) {
    print("iOS >= 9.0.0")
}

let os = ProcessInfo().operatingSystemVersion
switch (os.majorVersion, os.minorVersion, os.patchVersion) {
case (8, 0, _):
    print("iOS >= 8.0.0, < 8.1.0")
case (8, _, _):
    print("iOS >= 8.1.0, < 9.0")
case (9, _, _):
    print("iOS >= 9.0.0")
default:
    print("iOS >= 10.0.0")
}
```

But try not to. In the same category I would mention other similar mechanisms, such as `if #available(iOS 9, *) {}` or `[[NSProcessInfo processInfo] operatingSystemVersion]` or `[[UIDevice currentDevice] systemVersion]`. Just don't.

2.6 Selfish Navigation I have seen many apps using a single controller, and performing all visual changes inside of that controller, using animations, lots of views, and a severe case of MVC (Massive View Controller.)

This is wrong. The whole idea is that every screen in your application, as designed by your UI and UX teams, must map to one and only one subclass of `UIViewController`.

Of course in the case of the iPad, if you are using popover controls, you are most probably going to have lots of small view controllers. In that case remember that view controllers can be nested.

And to animate the transitions between those view controllers, you can define your own custom transitions to make them look any way your designers want.

2.7 Offline Confusion Making applications that store data locally and also load data from the network can be tricky, so here goes a simple pattern that works for every application that has to load data locally and from the network as well:

1. Start your application by loading the UI using data from the local data storage. Do not block the user.
2. If no local data available, inform the user with a "blank" state screen.
3. While this happens, start async network operations and download the data.
4. Save the remote data locally.
5. Notify the UI to refresh itself.

If you use `NSCache` then step 3 will return immediately, but your view logic should not know anything about this fact. Follow the steps above and in that order.

3. Project Management Smells

- Invisible Documentation
- Template Abuse
- Folder Clusterfuck
- Cocoapods Galore
- Rogue Compiler Warnings
- User Discrimination
- Interface Builder Attack

- Framework Infection
- iOS Nostalgia
- Backend API Anarchy
- Chatty API
- Splash Screen

3.1 Invisible Documentation I somehow feel bad having to say this, but seriously, this one is a constant in the industry. Look at the following (wrong) excuses to *not* to write documentation:

- We are agile
- Documentation is always out of date
- We haven't budget for that
- Nobody reads it anyway
- Our code is self-documenting
- Real developers don't need documentation

I say **bullshit**. If you have not written documentation, you have not done your job. And this applies not only to code that we write for ourselves, it applies particularly for the code that we write for others.

I will confess that I myself I am sometimes late at writing documentation; if you look at my project SwiftMoment in Github, you will see that I have an open issue to write documentation. That is (among others) the reason why this library has not yet achieved version 1.0 (it will, soon.)

What must be documented?

- Public classes, methods, enums, properties, everything that could be exposed to the outer world.
- Storyboards and XIBs: where are the UIs of the view controller classes defined, and how?
- Xcode schemes.
- Testing infrastructure setup (Jenkins, Xcode bots, etc.)
- Build setup procedures, including deployment information to the App Store or other signing issues, and how to solve them.
- Hidden dependencies: libraries, operating system versions, etc.
- Code that has been cut-and-pasted into the application, citing sources (URLs, books, etc.)
- Notifications and KVO use.
- Workarounds for bugs in external libraries of pods.
- Branching strategy for the repository.

How should these things be documented? The acronym representing the three pillars of code documentation is "ARW":

- API documentation in the code.
- README file at the root of the project.
- WIKI with more information, as required.

What should not be documented?

- Private members and classes.
- External libraries and pods.

Regarding `NSNotification` and KVO:

Notifications are great. They allows to decouple classes from one another and to reuse them in other contexts. They bring great power.

However, with great power comes great responsibility. Common problems I have seen around notifications are the following:

First, many developers still use plain strings for the notification names, scattered all over the source code. No. Use `static` constants, `extern` constants or Swift `enum` inheriting from String. In Swift embed the enum of notifications exposed by a class inside that same class, to increase code readability.

Second, no API help for the notifications whatsoever. This is a major no-no. Notifications are all about side effects; they introduce hidden dependencies in your code, that is, a change in one class may trigger a side effect on another. They can lead to code that is difficult to debug (hello thousand breakpoints scattered all over the code) and it makes the code really hard to maintain.

Every notification in the project should have its corresponding API documentation, explaining two basic things:

1. Who usually registers for this notification
2. When is the notification triggered and by whom

As general advice, remember that notification handlers should not trigger other notifications (that is, there not should be second- or third-order side effects) and that KVO notifications are the trickiest to figure out.

Also, I suggest that you start using `#keyPath()` in Swift 3, so that you have strong-typing compiler support for KVO observers.

3.2 Template Abuse Because we are "agile" it does not mean that we do not think anymore. For some reason we open Xcode and start pouring code even before sitting down and trying to understand the problem we are dealing with. And these days, "design" rhymes with UX and UI but nothing else.

Us, software developers, before starting to code a project, we must take a step back and think. We have to take a look at the available technologies and actually design our software using methodologies that actually make sense and convey information – and also, in the long run, become part of the documentation of the system we're trying to create.

Have you heard of CRC cards? Everybody has its mouth full of agile this and agile that but the truth is that apart from the daily meeting in the morning we barely apply any other agile technique whatsoever.

Before writing the next class in your project, take a time to figure out what every class, struct, enum and lambda in your project is there for, figure out how to test each part of the system, and then move forward.

3.3 Folder Clusterfuck People: it is 2016 and I still see project folders like this.

This cannot be, and it does not have to be like this.

Organize your code in folders, any arrangement will do: maybe using the typical "Rails" structure of "Models / Controllers / Views / Helpers" or any other organisation, but please take the time to organize your code in an appropriate way.

By the way, not using image catalogs in 2016 is a shame. Your designer should be producing them for you, as a matter of fact, so teach them how to use Xcode so that they can create those catalogs for you. Your project will be much better in the future.

3.4 Cocoapods Galore Do you really need to have more than five (5) external Cocoapods in your project? Seriously, no. You do not need that many. Think about it. iOS these days brings lots of functionality that it makes up for most small libraries out there.

Pick five big projects that you really need, and then only use those pods. For example, PromiseKit, AFNetworking, OCMock and frameworks like that. They really bring value, they really help you because they offer things that iOS still does not. All the other libraries (yes, including SwiftMoment) you do not need it.

3.5 Rogue Compiler Warnings This is a favorite of mine, and it keeps happening time and again. Particularly in old codebases that have not been updated, say, since iOS 5 and such.

Leaving warnings unattended is a big problem. Just assume that compilers are made by people who know the language much better than you, and in particular, they know how the language will evolve in the future.

Whenever you see a compiler warning, particularly when dealing with Objective-C code, you should immediately fix it. All code checked in source control should always: compile, pass tests, and have zero warnings.

In particular, Xcode 8 brings a new kind of warning to the table: "Runtime warnings", that is, situations that arise while the application is running under the supervision of Xcode. These could be harmless situation, such as autolayout incoherencies, but it could also be more complex things like memory leaks or circular object references.

A warning today, an error tomorrow. It is as easy as that. Pay attention to all of them and correct them.

3.6 User Discrimination Developers: stop releasing applications that do not include support for the incredible accessibility features of iOS!

Not only do they bring good karma points to your team and product, they are also required for UI testing in Xcode. So, please, make sure that you do support at least the following accessibility features:

- Dynamic text (so that the text in your UI changes size automatically depending on the preferences of the user.)
- Accessibility hints (used by VoiceOver to "speak out" different elements in the user interface for visually impaired users.)

3.7 Interface Builder Attack Storyboards are an interesting addition to the toolbox of iOS developers, however they bring a couple of problems to the table:

- Just like XIB files, they do not play well with source control. Have you tried fixing conflicts in Storyboard files? Not pretty.
- Impossibility to know how to reuse a controller in code. Should I `ViewController()` it or should I call `storyboard.instantiateViewController(withIdenfifier:_)`? And which storyboard should I use?

There are a couple of things that you can do to solve these problems:

1. Document the storyboards. Create a small `STORYBOARD.txt` file in your project that explains, for each storyboard file, which controllers are defined therein and what are the types of segues that connect them. Create entries for each storyboard in your application. Also: explain in the API documentation for each controller the name of the storyboard or XIB file where its UI is defined. This will make everyone's life easier.
2. Prefer individual XIB files for the UI of a particular view controller. Use the storyboards just for the navigation, but define the UI of an individual controller inside of its own XIB file. This way your controller will be more reusable in other contexts. Also, remember not to hard-wire references to other controllers in your own controller; this way you can make your controller more reusable.
3. Have separate Storyboards for iPad and iPhone. Enough said.
4. Prefer code to storyboards for large projects. Seriously. Believe me. And if you do not believe me, know that Google explicitly bans the use of Storyboards in their own iOS teams, and they do everything by code.

3.8 Framework Infection Since iOS 8 we have been able to create dynamic libraries for our projects. This is great, but at the same time I have started seeing a worring pattern: some projects divide their code in so many subframeworks, which is often an exaggeration.

This is unfortunate, because it introduces many small problems in terms of configuration and signing, and you should limit the number of subframeworks to a minimum. In particular, this technique is really useful only if you are targeting several different platforms with your code, and you want to share your code among them: tvOS, watchOS, macOS…

Otherwise, just use a good class design technique, create a flexible architecture, and just create a monolithic binary. Your build process will be simpler, and you will move as fast.

3.9 iOS Nostalgia iOS as a platform is evolving at a very, very high speed, and it has been like this for the past 8 years, and it will be like this for the forseeable future.

I know that some customers, for some reason, will insist that your code supports iOS 6 or 7 these days, but you have to tell them to stop doing that, for many reasons:

- Old versions of iOS are no longer supported and might have security problems.
- The costs of development and testing of old versions of an application are very high: you need a separate machine, old devices with old versions of the operating system, etc.
- It is very complicated to find documentation and online help to solve bugs in old versions of iOS.

The idea here is that you only support "current version - 1" at every single step of the way. This means that you have to write this down in the support documents that you provide to your customers, and that you have to start budgeting in September of every year the money required to migrate the code to the new version of iOS between June and September of the following year.

You can teach your customers that this is very important, and it will make life easier for everyone.

3.10 Backend API Anarchy This is a very important refactoring to make: if your mobile team is not in control of the backend API, then it is not in control of the mobile application at all.

The best mobile teams I know have a couple of good backend guys, usually building a proxy server between their mobile app and the backend provided by the customer. This brings many advantages:

- **Performance:** teams can tune the performance of the API that the mobile app uses, including the payload, its compression and other factors. You can also cache data, perform asynchronous refresh sessions, and deal with slow existing backends using SOAP or other clumsy RPC methods.

- **Quality:** you cannot rely in an API that is not in your control. What happens if your client inadvertently breaks the backend? Using your own proxy you can serve cached data, or at least fail gracefully.
- **Security:** you can ensure that all the communications between your mobile app and the API are encrypted and secure, using the industry's best practices, regardless of what the clients' IT team thinks.

APIs should not be chatty. To conserve battery power it is better to coalesce network operations in one call, returning all required information for a single screen in one simple step.

Also, explore new avenues using GraphQL and Socket.io, in order to have better APIs.

Final Recommendations

As you can see, this talk about refactoring did not only cared about your code, but also about the infrastructure around your code, and even your own team organisation. Good, maintainable code is the result of many factors, and so I leave here a couple more suggestions for your convenience, to make sure that your application code stands the test of time.

- Always use the latest Xcode and Swift
- Use a git branching model
- Migrate your projects for the future (iOS 11, etc)
- Plan accordingly
- Learn about classic design patterns
- KISS principle. Always choose the simplest solution that gets the job done.

Thank you so much for your attention.

Slides

Video