Cocoa is the new Carbon: the Future of Apple's Beloved Framework

Adrian Kosmaczewski

2015-02-26

In this talk, Adrian will provide lots of speculation and highly arguable unverified gossip, about how the design of Swift will lead Apple to redesign Cocoa into new directions, and maybe replacing it altogether. Some code examples will help him defend a point that nobody outside of Cupertino can sustain for sure.

Introduction

A wise man called Heraclitus said 2500 years ago that "Change is the only constant." He said it in greek, obviously, so actually he said these words: "

.” But I just do not know how to pronounce this, so I’ll leave it there.

The point is that Apple loves change. And to begin my talk I will go quickly back in time to 1976, to 1984, to 1995, to the two thousands, and back to now. In the meantime I will buy as much Apple stock as I can. You never know.

Look at the progression. We’ve gone from very humble machines to state-of-the-art computers, some of them handheld and operating a social and technological revolution. Soon, watches are going to appear in the horizon to complete this lineup, too.

Look at their CPUs; Apple has gone from a humble MOS 6502 to Gigaherz-fast ARM chips, eating very little power and providing enough calculation power to put a man on the moon, or, you know, to publish a selfie in Instagram.

At the same time, we have evolved from 8 bit to 64 bit architectures, forcing us to rewrite lots of low-level pointer-arithmetic code, because you know, that’s where the fun is.

And finally, the most important part of the evolution for us, software developers, is the constant change of “official” programming languages; from Wozniak’s Integer BASIC in 77 to Swift, Apple has constantly required us to migrate our skills to newer, safer, and buggier programming languages.

This leads me to another clear evolutionary path, if you look at this slide closely: we have gone from a procedural world, the one of BASIC and Pascal, to an object-oriented one, with C++ and Objective-C, and Swift is opening the way for a more functional way of doing things.

Of course, every time the marketing department said the same thing about those languages; that they are the best, the fastest, the blah blah blah. So, in a sense, we can say that “plus ça change, plus c’est la même chose.” Clearly, Jean-Baptiste and Heraclitus never agreed on the subject of change.

Transitional Technologies

Another thing that has not changed is that Apple always provided us with what I call “Transitional Technologies.” That is, emulators, compatibility layers and binary formats that have allowed us to move our applications from one architecture or language to the next, and often to run it without changes in the new platform. The history of Apple is quite interesting in this respect.

The earliest example is the “Mac 68k Emulator Layer,” bundled with Power Macs back in the 90’s, so that old Mac applications compiled for the Motorola 68000 CPUs could run. Remember that? This layer existed even in Classic, yet another compatibility layer that allowed OS 9 applications to run on OS X… but I digress.

Then we have the Macintosh Application Environment. Anyone remember this one? Back in the days where we were waiting for Jobs to return, when Copland, Taligent and OpenDoc were still worthy of attention, back in those days Apple released a compatibility layer for Sun OS Unix boxes with SPARC chips, where Mac apps could run. I guess I do not need to say why nobody remembers it.

More recently, some of us remember the "Classic Environment," allowing users to run their old OS 9 apps in Cheetah, Jaguar and Tiger.

Going back to the compatibility layers, in the 2000's Apple introduced "Carbon." It was a sibling framework to Cocoa, deprecated in Mountain Lion, never ported to 64 bits, allowing OS 9 applications to be recompiled as native OS X apps, even if the look and feel was not exactly the same, but at least you would get Aqua buttons, which is everything users wanted back in 2002.

Carbon was Apple's transitional API, the one that enabled OS 9 applications written in C and C++ to be compiled as native OS X applications, waiting for developers to rewrite these applications as native Cocoa apps in Objective-C. Something that the Photoshop team took ages to do, but well, that's another story.

In the meantime, preparing for even more changes, Apple started two famous very open source projects; the first is WebKit, and I think everybody agrees on how important it has been for the web. But more important for us tonight is LLVM, a project that initially started as a way to modernise the old and rusty GCC compiler infrastructure, and Swift can be said a direct descendant of LLVM.

More or less by that time, Apple decided to use Intel chips instead of PowerPC ones, and so it introduced two transitional technologies at the same time: first "Rosetta," an emulator layer which allowed OS X PowerPC apps to run on Intel chips, until their developers would recompile them as native i386 ones. After recompiling them, for a while Xcode would produce "Universal Binaries." They are basically NeXT fat binaries, merged with the "lipo" tool, containing both PowerPC and Intel code; applications built using this technology could run seamlessly in older and newer Macs, between 2006 and 2011.

Based on the new LLVM compiler infrastructure I mentioned before, Objective-C evolved for the first time since 1988, and in 2006 Apple released Objective-C 2.0, which brought many new features:

- Garbage collection
- Properties
- Fast enumeration
- 64-bit capabilities

LLVM is also the original technology that brought us ARC or the Xcode Static Analyzer; in particular, ARC is a fundamental piece for Swift, providing compile-time memory management, instead of relying on a potentially heavy garbage collected runtime.

Cocoa

Carbon was introduced in 2000, and deprecated in 2012, so, my question is, maybe Cocoa is the new Carbon? Is Cocoa the new transitional API that will enable Objective-C applications to run in OS X and iOS while developers rewrite them in Swift, while we wait for Apple to release the… what? The Swift "Birdy" Framework?

I argue that we are back again in a transitional period. Objective-C will most probably only be updated as long as Swift requires it; the new "nullable" pointer annotations introduced recently are a clear indication of that. Objective-C is evolving into a language that supports future Swift libraries. New hardware like the Watch will come and will require new software.

My theory is that Cocoa will be around for the next 10 years. That is right; Cocoa and Objective-C will be around until 2025, and during this time, of course, Swift will become the de facto programming language for writing Mac, iOS and Watch applications. And the demise of Objective-C will surely make some developers very happy.

The scenario is all set for yet another migration. And Cocoa is the compatibility layer this time.

Inspiration

With all of this historic background, let's take a closer look at Swift. What is Swift? Apple says that the language can be defined as "Modern, Safe, Fast and Interoperable." For the moment I can say that Swift is a great language to create strongly-typed "Hello World" applications. But I'm probably too grumpy, so don't take my word for it.

One of the best explanations comes from Twitter: "Swift is Haskell with C# syntax." In my personal experience with the language, beyond cursing the pitiful state of the developer tools, I came to the conclusion that Swift is *a hybrid language compatible with both object-oriented and functional APIs.* That is what I call Swift. You might disagree with this definition, and we could talk it over a beer after this talk, but in essence this is how I see the language right now.

Chris Lattner, the creator of Swift, says that one of the foundational bricks of the language are protocols; he has not really mentioned the functional part as one of those bricks, yet the developer community has a different opinion.

All of this does not tell us much about that "Birdy" framework of 2019; at best, I think we can just grab some inspiration from other languages and their associated frameworks. What features of the language would influence the design of this master framework?

- Protocols
- Strong typing

- Optionals
- Generics
- Tuples
- Enums

Furthermore, what could we borrow from other libraries or frameworks, to imagine the next version of Cocoa? Let's review quickly some popular programming languages, and see what could be borrowed from each.

From C

I think that LINQ is a great example of something that could and probably should be ported to Swift.

Oh, and by the way, maybe Xcode could take a hint or two from Visual Studio, if you ask me.

From JavaScript

There are great examples of short, sweet JavaScript libraries around, most of them heavily developed during the past 10 years, after the explosion of AJAX. For example jQuery, maybe the quintessential JavaScript framework; small, it bridges compatibility problems between browsers, it uses JavaScript's functional features extensively, and it has inspired a whole industry of JavaScript frameworks such as Zepto, Sencha Touch, YUI, Ext.js, Underscore, and the list goes on.

Actually I do expect somebody to come up with a Node.js-like web server written in Swift; it's only a matter of time.

From Haskell and Erlang

What should we take from these? Anything but the syntax. I think we got that, as a matter of fact; so the key elements we could adapt are actors, parallel execution and fail-safe features, many of which are already available in the language, to be fair.

From Ruby and Rails, RSpec, Cucumber...

We can borrow the great DSLs built on top of it; and I look in particular to CocoaPods, which is simply a DSL based in Ruby that help us handle external dependencies in our projects. Could not we just use Swift to do the same? I think we could and we should.

The fact that Swift allows "last parameter" lambdas to be called without parenthesis is very similar to the mechanism enabled by the "yield" keyword in Ruby.

From C++

One could argue that the current Swift standard library borrows some functions from the C++ standard library, and maybe also a bit from Boost. Although generics are not quite the same thing as C++ templates, in spite of the similar syntax, we can use them to compose objects in very interesting directions.

I would like to see who will be the Alexandrescu of Swift, providing us with a full book of crazy uses of generics, redefining what we can do with them, including enums, structs and not only classes in the mix.

From Java and Scala

We could take IntelliJ's AppCode (thankfully!) and then lots of examples of how *not* to do things: Eclipse, Maven, Struts, the low speed of compilation and execution, and I'm pretty sure you have plenty of other examples.

Let's not repeat those mistakes, please.

From Scala, of course, we have an uncannily similar syntax, one that stops there, of course, because the execution models of both languages are quite different, as a matter of fact.

Gedankenexperiment

So, the question is, what would Apple's official Swift framework include? What would it look like?

I will cheat; in this talk I will see the current state of Open Source development in Swift to provide us with some hints. Maybe Apple is also paying attention to these projects, and will give us APIs looking and behaving like the ones I'm going to describe now.

I was going to show some live coding demos, but frankly the current state of playgrounds and the Swift compiler in general played against me tonight. So I will stick with very simple code samples right on the slides.

Swift Standard Library

Of course the first example to pay attention to is Swift's own standard library. I cannot avoid looking at it and thinking that it is not yet ready; that it is essentially incomplete, yet it is probably the embryo of the next evolution of Cocoa.

The most important observation I can make from typical code examples using the standard library is how few classes you see in use; most of the APIs are functions operating on structs; even better, these structs can be made immutable, with all the benefits that brings.

So, few classes. You know a Swift developer has issues when you spot a static class method somewhere during a code review.

On the negative side, one thing that the Standard Library needs, and fast, is a more complete documentation of it, provided and maintained from Apple. It is actually embarrassing that we have to rely on SwiftDoc instead.

LINQ in Swift

I have found a complete project in Github exploring the possibility of using Swift's standard library in a similar fashion to LINQ in C#. The slide shows a snippet of code in C#.

But to be fair, being able to query data structures in memory using a SQL-like language is something that Cocoa has been able to do since NSPredicate was introduced, back in the early 2000, together with Core Data; NSPredicate itself is heavily dependent upon KVO, another technology available since the 80's in Cocoa.

However, neither NSPredicate nor KVO provide the compile-time verification power of C#, not even remotely. Could we use Swift, which comes bundled with such power, to evolve those technologies into a LINQ-like library? I think that the answer is a resounding yes.

Dollar.swift

One Swift library that has been heavily inspired by jQuery and Underscore is "Dollar.swift" for example. It provides several functional constructions, allowing developers to operate directly on functions, compose and reuse them as required. This library makes heavy use of everything functional that Swift has to offer.

Of course, Swift and JavaScript have dramatically different runtime differences; JavaScript runs in a single-threaded environment, and for performance reasons, JavaScript frameworks tend to be asynchronous, delegating behaviour to "time-out'd" functions, in order not to block the browser's run loop.

But just as JavaScript, Swift is a hybrid, "kinda-functional" language, with a syntax that allows for similar constructions. Let's imagine for a second a library such as Socket.io, but written in Swift? The mind boggles.

Alamofire for networking

This is probably one of the most interesting libraries available for Swift today, and I'm pretty sure that everyone will use it. It has very interesting characteristics, showing how to build asynchronous APIs in Swift.

Look at this code; we have a very "JavaScript" like syntax. We specify the request that we want to send to the server by chaining a couple of functions after the others. The parameters are simple, and some anonymous functions are used as asynchronous callback methods.

In Swift, methods are actually curried functions; not only that, but functions map directly to Objective-C blocks, and as such they can be used in the same situations, with the same idioms.

NSNotifications for Architecture

There are few components so loved and misused in Cocoa like the NSNotificationCenter. We all love it and hate it vehemently at the same time. So it is not a surprise that both Mike Ash and Chris Eidhof have both thought of replacing it with a more "Swift-like" API, and in this case I will present the one shown by Chris.

As you all know, and if you don't now you do, Chris is one of the creators of objc.io, the greatest online magazine about Cocoa there ever was. He is also one of the co-authors of the book "Functional Programming in Swift" so I guess he knows what he's talking about.

So that's why one day he presented a replacement for the Notification Center, and it looks like this.

- Generics for notification definitions
- Lambdas as callbacks

Swift Events for communication

Events are quintessential in UI toolkits; they allow developers to receive asynchronous information about the activities of users. But an event is essentially an object that wraps some code that has to be executed at some point in the future; so a very simple, almost naïve implementation of events could be the one shown in this slide.

To use this API, just assign some handlers to an event, and use the `raise` method to call them all at once. Simple.

Pay attention to a very interesting feature of Swift! Methods that take multiple parameters are the same as methods that take tuples as input. This makes the syntax used to raise events much simpler, straightforward and easier to read.

Swifty User Defaults for storage

This library is a perfect example of small additions on top of NSUserDefaults, in order to have a more "Swift-like" API. The author uses subscripts to enable a simpler syntax for getters and setters using subscripts, as well as cleverly using generics to provide a strongly-typed interface.

Swift Moment for time manipulation

Finally, without shame, I'd like to talk about my own take on a pure Swifty implementation of a date & time library. Swift Moment was recently featured by Natasha Murashev and Dave Verwer in their respective newsletters, and it has been heavily inspired by a JavaScript library called Moment.js, quite popular in web development circles.

Swift Moment is based in several typical Swift paradigms.

- The most important being the heavy use of functions and structs instead of classes and methods. I have made the early choice of making the library as similar as possible to Moment.js, and as such I use global functions, such as "moment()" to create new instances of a "Moment" struct. This structure encapsulates an NSDate instance, and it provides an easier API to typical date and time operations, such as conversion to strings, getting components such as day of the week or others, and many other operations.
- Another thing that I've extensively used in Swift Moment are operators. Being able to say whether a date is older than another, or similar, are cases where operators work beautifully well.
- Finally, I've tried as much as possible to make immutable structs out of Moment and Duration. It makes sense; they represent fixed points in time, or fixed time spans, and modifying them at runtime makes no sense.

When using Swift Moment, developers have a straighforward interface. Ambiguity is almost nonexistent.

Conclusion

I cannot claim, by any means, to know what is going on inside of Cupertino as we speak. I think, however, that Swift will move the development of Cocoa into new directions, thanks to the many features of Swift:

- Protocols, even on enums and structs
- Pattern Matching
- Functions chaining
- Function currying
- Subscripts
- Functions and structs instead of classes and methods.
- Custom operators, where it make sense.
- Immutability
- Optionals
- Tuples
- Generics

See the pattern? There are not so many objects or classes. APIs are strongly typed. Optionals make sense. Data is immutable. Function composition takes over object composition. Pattern matching and tuples here and there. Lambdas passed around as first-class citizens. Swift has a tremendous potential in terms of syntax and expressive power, and I frankly think that we have not seen anything yet.

On the minus side, I have to say that I would like to see more reflection capabilities in Swift; it is something that C# and .NET offer, and heck, even Objective-C had it, and we've lost it. Powerful frameworks have reflection.

In any case, I think that we are moving towards a functional world, and as such I recommend everyone – including myself – to learn more about the Functional Programming Paradigm, in order to be a better post-Cocoa developer, thanks to Swift. And Junior will talk about that right after this talk.

However, it is not very clear to me what will be the next step regarding user interfaces. We all know that there is a tension between UI development and functional programming, because the former is all about state, and the latter frowns upon it. Will we see a functional-based UI component library? Does this make any sense? Or will UIKit (and maybe UXKit) simply change to support Swift paradigms?

I guess, time will tell. Thanks a lot for your attention.

Special thanks to Fabrice Truillot, who has been writing Mac applications for 27 years in many of the languages I mentioned, for reviewing drafts of this speech and providing great feedback.

Slides