Extracted from:

Xcode Treasures

Master the Tools to Design, Build, and Distribute Great Apps

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Xcode Treasures

Master the Tools to Design, Build, and Distribute Great Apps

Chris Adamson
edited by Tammy Coron

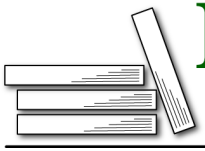
Xcode Treasures

Master the Tools to Design, Build, and Distribute Great Apps

Chris Adamson

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Development Editor: Tammy Coron
Copy Editor: Jasmine Kwityn
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-586-3
Book version: P1.0—October 2018

Introduction

A good craftsperson doesn't blame their tools… but you wouldn't know this from the way some developers talk.

Hop on Twitter or stop by your local CocoaHeads meeting, and someone's sure to be complaining about *something*. A minor annoyance here, a show-stopper there, a grudge from years ago that needs to be nursed now… it all leads to a constant sense of negativity about the work we do as developers.

A lot of that griping is directed at Xcode, the tool that's more or less indispensable when developing for Apple platforms. Just search for `xcode` on Twitter and you'll see screeds and tweetstorms damning Xcode for every sin a developer tool can commit—as if the app had been developed by the Marquis de Sade himself, for the sole purpose of torturing iOS developers.

But… come closer… let me blow your mind for a minute.

Xcode is good, actually.

Been There, Done That

As a longtime developer, I can be pretty jaded about things. But when it comes to developer tools, it's not the tools I've grown weary of, it's the complaining.

Before getting into Apple development, I used to edit a developer website for Sun Microsystems and O'Reilly Media called `java.net`. This was a site with feature articles, news items, and a repository of open source projects. Because Sun's stuff was on there, we naturally did a lot of coverage of their IDE, which was called NetBeans. But within the Java community at the time, there was a holy war between NetBeans and another IDE named Eclipse.

The Eclipse fans were not always diplomatic in their distaste for NetBeans, which is a diplomatic way of saying they were vulgar, toxic zealots about it. They didn't just dislike NetBeans, they *hated* it. They hated the application, hated the people who worked on it, and hated anyone who said they liked it. They made everything personal. It wasn't about the difference between

lightweight and heavyweight rendering of UI components, it was about US VERSUS THEM.

But over the years, a funny thing happened. Android hit the scene and initially adopted Eclipse as its IDE of choice. Yet, in recent years Google has pushed it aside in favor of a new IDE, "Android Studio." And all of a sudden, I see developers on that side of the fence saying how great Studio is compared to Eclipse. And how Eclipse sucks. And how Eclipse has *always* sucked.

To which I'm like… "Really? Because you pretty much used to say that I'd be committing professional malpractice if I *didn't* use Eclipse."

This was the point at which I officially stopped listening to the mob when it comes to the relative merits of IDEs.

What's in an IDE?

I keep using that acronym: IDE. Let's stop for a second and look at what we're even talking about. An IDE is an *integrated development environment*. In other words, it brings together multiple tools for developers and integrates them so they work well together, often all within a single application.

For the most part, you don't actually have to use an IDE at all. Look at what goes into an iOS or macOS app:

- You edit source code.
- You build UIs with a visual editor, although you can also build UIs in code.
- You compile the source into executable code, and you link it with libraries and frameworks.
- You collect the executables, along with resource files like graphics, sound, localizations, metadata, etc., into an app bundle, which is just a folder that adheres to a known format.
- You use a code-signing tool to cryptographically prove your identity as the creator of the app.
- You distribute this to users or upload it to Apple for review and publication on their App Store.

The thing is, pretty much every one of these steps can be performed individually, with separate applications. You can write your source in any number of popular text editors, compile it on the command line with `clang` and `swiftc`, use scripts to assemble the bundle, and use Apple's `codesign` tool to sign the

result and ready it for Gatekeeper or the App Store. Then you upload it with a browser to submit it for Apple's review.

And yet, almost nobody does this. You can own your workflow, but it's way too much work for way too little benefit.

In fact, consider the unappreciated benefits of combining these into a single application:

- When the UI builder and the source editor are in the same app, the UI builder can insert code to create your IBActions and IBOutlets for you.
- When the compiler gets a live look at the code, it can offer accurate auto-completion as you type, not based merely on the matching string fragments, but by figuring out what will and will not compile at your insertion point. As a bonus, the compiler can also show you warnings and errors as you type.
- To top it off, the code-signing process, which used to be a manual mess of tools, can be made automatic and actually do the right thing… most of the time, anyway.

This isn't just more convenient, it's more powerful, in substantial and interesting ways.

No Regrets

Now think about how you use Xcode. There are a ton of menu items you've never used, right? What about those project settings that get filled in for you? Have you ever really given them a second look? The scheme selector that's never been anything more than a way to switch simulators for build-and-run; what more can it do? Those weird little inspectors all over the right pane in storyboards; what's their purpose?

Isn't it possible there's a whole lot of power to this app that you've never used, because you didn't know it was there, and never needed to look for it?

Isn't it possible that at some point you've said "I wish Xcode did *X*", and that maybe Xcode actually *does* do *X*, and you just didn't know?

That's the purpose of this book: to go beyond the basics; to take an in-depth look at the many different areas of Xcode and show you the neat stuff you may have missed. It's a collection of handy tricks, clever customizations, and power techniques that will leave you wondering how you ever overlooked them or lived without them.

About This Book

This is a book about the tool Xcode, not about the software development kits (SDKs) for the various Apple platforms. That means you should *not* make this your first book for learning iOS or macOS programming. This book is for developers who know the SDKs, at least a little, and want to get better at making apps.

Notice that the preceding paragraph said SDKs... plural. Xcode supports development for four platforms—iOS, macOS, tvOS, and watchOS—and this book embraces that diversity. You'll find examples for all the Apple platforms, most obviously in Chapter 11, *Platform Specifics*, on page ?, but also wherever it makes the most sense for the material. For example, in Chapter 2, *Storyboards: Appearance*, on page ?, the examples use Mac storyboards to work with Auto Layout because it's easier to see layout changes by resizing windows than by running on differently sized iPhone simulators. Don't worry; Apple platforms share so much code in their SDKs, it's not that hard for an iOS developer to get the hang of macOS, and vice versa.

Similarly, the book will use whatever Xcode-supported programming language best illustrates the material. While Swift is the default for most examples, some will use Objective-C or C, as is the case in Chapter 6, *Debugging Code*, on page ? when inspecting memory buffers directly in the debugger, or catching errors that are only possible in those languages.

Each chapter takes on a more or less distinct topic, with little or no dependency on earlier material, so feel free to hop to whatever catches your interest from the table of contents. You don't have to read the whole thing in order if you don't want to.

Expectations and Technical Requirements

This book assumes you have some familiarity with developing for one or more of the Apple platforms: macOS, iOS, tvOS, or watchOS. Presumably, you've at least finished an introductory book; if not, may I recommend *iOS 10 SDK Development [Cla17]* to learn the iOS SDK. If you'd also like to take a deep dive into the Swift language itself, in a more platform-agnostic form, take a look at *A Swift Kickstart [Ste17]*.

The book uses the current version of Xcode, which at the time of this writing is Xcode 9. Historically, Xcode only runs on the current version of macOS

(and for a limited time, on the previous version as well), which at the time of this writing is macOS High Sierra (10.13).

Online Resources

You may find the code for this book on the book's Pragmatic Bookshelf website.[1] If you find a problem with the text, please report it using the errata submission form.

1. https://pragprog.com/book/caxcode