The Missing Manual for Swift Development

Written by Bart Jacobs

Cocoacasts

The Missing Manual for Swift Development

Bart Jacobs

This book is for sale at http://leanpub.com/the-missing-manual-for-swift-development

This version was published on 2017-09-27

* * * * *

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

* * * * *

Table of Contents

Welcome

Part 1: Foundation

1 Your Foundation

What Is Your Foundation

There Is No Clear Path

2 Where to Start

Setting Yourself Up for Success

A Path Without Distractions

Choose Your Teacher Wisely

3 Know Your Tools

Xcode

Developer Portal

Provisioning

Instruments

Command Line

Third Party Tools

4 Adopt Best Practices

What Are Best Practices

Patterns

Anti-patterns

Good Practices for Swift

Be Critical

5 Security

Make It Hard

Plain Text

Obfuscating Information

Fetching Sensitive Information

Encryption

Privacy

Logging and Debugging

Educating Your Client

6 Don’t Ignore Your Foundation

Under the Hood

Dependencies

Don’t Ignore the Fundamentals

7 Respect the SDK

Know Your Limitations

Respect the SDK

9 Design Patterns

Model-View-Controller

Model-View-ViewModel
Singletons
Dependency Injection
References, Delegation, and Notifications
Master These Patterns
6 Protocol-Oriented Programming
7 Reactive Programming

Part 2: Swift
1 Learn Swift With an Open Mind
Reference Types and Value Types
Protocol-Oriented Programming
Type Safety
Best Practices
Forget What You Know
2 Be Critical
3 Embracing Optionals
Warning Sign
Elegance and Beauty
Swift to the Rescue
Embrace Optionals
4 Mind the Exclamation Mark
Use Cases
Convenience and Laziness
Don’t Be Lazy
When I Use the Exclamation Mark
A Personal Choice
5 Exclamation Marks and Fatal Errors
Clarity Over Subtleness
Choosing for Clarity
6 Smelly Code
Forced Unwrapping and Conversion
Monster Classes
Massive Methods
Ignoring Errors
Singletons
String Literals
Is Your Code Smelly
7 Value Types and Reference Types
What’s the Fuss
An Example
Benefits of Value Types
When to Use Value Types
8 Catching Errors
When Should You Handle Errors
Where Should You Handle Errors

Notifying the User
Monitoring Application Health
Don’t Ignore Them
9 Using Fatal Errors to Write Elegant Swift
What to Do When You Don’t Know What to Do
Guarding Against Unexpected Events
Use Fatal Errors Sparingly
Clarity and Elegance
Part 3: Projects
1 A Brand New Project
Step 1: Setting Up the Project
Step 2: Organizing the Project
Step 3: Adding a README.md
Step 4: Build and Run
Step 5: Adding a **.gitignore**
Step 6: Putting the Project Under Source Control
Step 7: Pushing the Project to GitHub
Step 8: Optional Steps
2 Project Structure
An Example
View Controllers
Test Target
On Disk
What Do I Gain
Tools
Caveats
3 Project Hygiene
Projects Evolve
Obsolete Files
Comments
Documentation
Don’t Commit Everything
Guidelines and Exceptions
4 Document Everything
Start With the Basics
Make It Easy
What to Document
Up to Date
Make It a Core Tool
I’m a Team of One
Make It a Habit
Time Is Money
Part 4: Workflow
1 Testing

- Where to Start
- Code Coverage
- Writing Better Code
- Dependency Injection
- Xcode and Testing
- My Current Test Setup
- What Are You Testing
- It Takes Time

2 Continuous Integration

- What Is It
- Why Is This Useful
- Long Hanging Fruit
- Avoid Human Tinkering
- TestFlight
- Painless Releases
- Make It Robust
- Transparency
- Learning Curve

3 Refactoring

- Technical Debt
- Building for the Future
- The Fallacy of Sunk Cost
- Starting Anew

4 Source Control

- The Basics
- Don’t Break These Rules
- Commits
- Stashing
- Patching
- Git Flow
- Some Tips

5 Dependencies

- Minimize Dependencies
- What Is a Dependency
- Fewer Dependencies
- Mapping the Liabilities of a Project
- Don’t Make Your Life Too Easy
- Rolling Your Own
- Choose Wisely
- Becoming a Better Developer
- Watch Out for the Defaults
- Challenge Yourself

6 Automation

- Scripting
- Build Phases
- Automated Testing
- Documentation
- Continuous Integration

fastlane
Keep It Simple
7 Privacy
Protecting the User's Privacy
Who Do You Work With
Giving the User Control
Third Party SDKs
Choose Wisely
8 What to Do When You Inherit a Software Project
First Things First
Build and Run
Collect Data
Dependencies
Working on the Project
Document Everything
And Beyond
9 Speed, Quality, and Technical Debt
Speed and Quality
Technical Debt
Focus
How to Get Rid of Technical Debt
Taking Shortcuts
Part 5: Team
1 Code Reviews
Just Start
Calendar and Agenda
Make Them Actionable
Keep Them Small
Be Prepared
Tools
That Hurts
Convincing Management
Frequency
But It's Just Me
Give It a Try
2 Adopt a Style Guide
Why Have One
Automation
Rough Transition
3 Working In a Team
Hire the Right People
Leadership
Communication
Give It Time
Respect
When Things Hit the Fan

Ownership and Responsibility
Share and Ask
4 Being a Leader
Listen and Be Open
Modesty, Humility, and Respect
Work as a Team
Working With People
Follow Your Gut
Lead
Part 6: Career
1 Open Source
Start Small
Taking the Plunge
Documentation
Taking It Seriously
Giving Up Control
Taking Responsibility
2 You Are the Constant in Your Career
Putting Yourself First
Setting Goals
Learning Requires an Investment
Take Care of Yourself
Looking Back
3 Build That Application
Build and Ship
Show What You Can
Stay Ahead
Rinse and Repeat
Build That Application
4 Protect Your Productivity
Working From Home
Working In an Office
Minimizing Interruptions
Find a Quiet Place
Deal With It
5 Building Your Portfolio
Scratch an Itch
Setting Goals
Learn the Basics
But Start Creating
6 How Badly Do You Want It
Contribute
Build
Maintain
Plan and Manage

Gain Experience
Be Yourself
How Badly Do You Want It
What About You
7 Freelancing and Subcontracting
Freelancing and Subcontracting
Being Picky
Firing Clients

Part 7: Products
1 Ship, Ship, Ship
It's Hard
How to Ship Consistently
Internal Deadlines
Remove Friction and Clutter
2 Talk to Your Customers
Solve a Problem
Ground Zero
Is This for You
But It's Fantastic
A Recipe for Success
3 What Is Stopping You From Shipping
Why Is This Important
Start Small
Remove Clutter
4 Motivation Will Get You Only Halfway
Motivation Won't Cut It
Running a Marathon
Pulling the Plug
Start and Persevere
Challenge Yourself
5 How to Make a Living as a Mobile Developer
Paid Up Front
Freemium
Advertising
In-App Purchases
Subscriptions
Donations
Finding the Right Business Model
Experiment But Be Smart
Evolution

Part 8: You
1 Being and Staying Productive
Maintaining Focus
Protect Your Productivity

Distraction Is Addictive
Take Care of Yourself
2 Stop Looking for the Silver Bullet
An Example
Investigate and Test
Stop Looking for the Silver Bullet
3 You Are Not an Imposter
What Is It
Why Do I Bring This Up
Curing Imposter Experience
Talk About It
It's Common
Don't Believe Everything You Read
Don't Let It Affect You
Success Through Failure

Part 9: Learning
1 Choose Your Teacher Wisely
Information Overload
Who to Trust
Focus, Focus, Focus
Never Stop Learning
2 Taking a Shortcut
Internship
Freelancing and Subcontracting
Learn, Learn, Learn
Build, Build, Build
Choose Wisely
3 Some Things Are Hard
Cut Yourself Some Slack
4 Learn the Rules, Then Break Them
Examples
Growing as a Developer
Creating Something Better

Welcome

The title of my book, **The Missing Manual for Swift Development**, accurately describes what I have in store for you. It's the guide I wish I had when I started out as a software developers years and years ago.

The Missing Manual for Swift Development is a summary of what I've learned over the years building software for Apple's ecosystem. Many of the lessons in my book I learned from experts in their field and, unfortunately, just as many I learned the hard way. I hope that some of the topics in this book can help you on your way to become remarkable in what you do. That's your goal. Is it not?

Writing a few lines of Swift is surprisingly easy. Once you start to dig deeper, though, you discover that building an application for Apple's ecosystem is more challenging than it seems. **The Missing Manual for Swift Development** outlines the challenges you face along your journey and how to overcome them.

Some of the more obvious topics I cover include dependency management, source control, code reviews, continuous integration, style guides, working in a team, tooling, project organization and documentation, and release strategies.

The topics I found most interesting to write about, however, are more meta, such as when to break the rules, freelancing and subcontracting, staying productive as a developer, shipping projects, leaving your comfort zone, and dealing with challenging problems.

My book doesn't include many code snippets or sample projects. The goal of this book is to provide insights and answers to questions that are often overlooked or ignored.

The Missing Manual for Swift Development is for every type of developer, but it primarily focuses on Swift and Cocoa development. If

you’re developing for Apple’s ecosystem, then you’ll find a lot of useful information, regardless of your experience.

There are very few shortcuts in software development. You pay a price for most of the shortcuts you read about, one way or another. But there are a handful of shortcuts that can speed up your learning and your career. A proper education is one of them.

I hope that my book helps you in some way, big or small. If it does, then let me know. I’d love to hear from you.

Enjoy,

Bart

Part 1: Foundation

1 Your Foundation

Developers that come to Apple's platforms have many different backgrounds, and some of us are better prepared than others. Regardless of your background, though, you can make sure that you're a good fit for Apple's ecosystem by investing in your foundation. That's the subject of this chapter.

This may sound boring at first, and for some it *is* boring, but it's essential if you plan to create robust, scalable applications and if you want to have a long career in the mobile space.

There's a slew of reasons why developers are drawn to Apple's ecosystem. Many of us make the jump because of the shiny stuff. ARKit and Core ML are good examples of what I like to refer to as shiny stuff. Both are amazing technologies, and I'm sure you've seen stunning examples of developers taking advantage of ARKit. ARKit and Core ML are the cherries on the cake. And the cake is your foundation. Without the cake, there are no cherries.

What Is Your Foundation

The foundation is what happens to interest me most. It isn't as compelling or visually stunning as ARKit, but it's essential for every single application. Not only is that why it's so important, but that's also why it's worth investing time in your foundation. It pays dividends for a long time. Let's start with the language.

Swift

As a Swift developer, the first component of your foundation should be the language you use day in day out. Swift. Since you're reading this, I'm going to assume that you already have a good grasp of the Swift language or, at a minimum, you've browsed The Swift Programming Language and you've written a handful of lines in a playground.

I'm sure I don't need to convince you of the importance of knowing and understanding the Swift language. How are you going to write a fantastic novel if you don't master the language the novel is written in? Why would this be any different for software development?

But this doesn't mean you need to spend days, weeks, or months learning the Swift language. I recommend a different approach that's more fun and more practical. I don't like spending hours or days learning a new library, framework, or API. I learn best by applying what I read and learn. The same is true for the Swift language.

Start with the basics and use it. You can pick up the basics of any language in no time. But, as with any language, you only *get* what you've learned by applying it. That means writing Swift. You can use Swift in a playground or start with a simple Cocoa application. Apple did a fantastic job by introducing playgrounds alongside the unveiling of the Swift language. Use them. Launch Xcode, create a playground, and play with Swift.

Optionals

Many developers will go through a phase in which they struggle with the language. That's fine. Let yourself be frustrated for a while. It's natural, and it means that you're noticing aspects of Swift that are core to the language. Be skeptical and ask questions.

Based on conversations with students and readers, optionals are one of the most common obstacles developers need to overcome. I genuinely love optionals, but I too was frustrated when they first crossed my path. This wasn't surprising considering my background in Objective-C. As you probably know, Objective-C isn't as strict about `nil` pointers. Dealing with `nil` is an entirely different story when you're writing Swift.

If you're still struggling with optionals or don't see how they fit in the language, then I need to ask you to trust the language for now. It will click at some point, but you need to embrace optionals. Don't use the exclamation mark to plow your way through a forest of optionals. Optionals are an integral part of the language and an essential component of one of Swift's key features, safety.

Step Up Your Game

The more you use the language, the more you start to appreciate it. What I tend to do is gradually add more complexity to the code I write. Learn about value types and reference types. Discover what protocols are and why they're useful. You may want to explore generics at some point. Take a close look at enumerations and find out why they're so much more powerful than their Objective-C counterparts.

If you think you know the Swift language, then I recommend taking another look at The Swift Programming Language. Every time I read Apple's guide, I learn something new. Most of the times, I can directly apply it to the code I write day to day. I love it. Swift was designed to be more expressive, and it feels richer than Objective-C. That's something you can and should take advantage of in the code you write.

Concepts

Even if you still use Objective-C, there are concepts that you need to know about, regardless of the language you use. These concepts are often scary to new developers, such as memory management, asynchronous programming, and grand central dispatch. These are concepts and technologies that are often ignored or overlooked. Unfortunately, this leads to memory leaks, crashes, and a lot of frustration.

This frustration often leaves a bad taste in your mouth, and it can even affect your fondness for the language and the platform. There's nothing as frustrating as debugging an issue caused by something you don't fully understand. It happened to me many times, and it simply isn't fun. It pays off to invest time in learning the ins and outs of these fundamentals. I talk more about this later in the book.

SDK

You can do quite a bit with the Swift language that doesn't involve Apple. The language is available on Linux, and there are several frameworks that you can use to build web APIs powered by Swift. There are also a number of tools and libraries you can use to write scripts using Swift, such as Marathon, Rainbow, and Commander.

But most of us use Swift to power mobile or desktop applications, which means we need to become familiar with a bunch of tools, frameworks, and libraries. The most common ones are **Foundation**, the bread and butter of every Cocoa developer, **UIKit** and **AppKit**, and several other ones, such as Core Data, AVFoundation, CloudKit, and HealthKit. The time you spend exploring and learning about these frameworks and libraries is time well spent.

Design Patterns

There are dozens and dozens of design patterns you can pick and choose from and a handful of these design patterns are used by the frameworks and libraries you use on a daily basis. At a minimum, you need to know about and become familiar with these fundamental design patterns. Let me give you a quick overview of the most important ones. Later in this book, I zoom in on a few of these design patterns.

Model-View-Controller

The first design pattern you inevitably come into contact with when you create your first project in Xcode is the **Model-View-Controller** pattern. It's this pattern that powers most Cocoa applications. If you've been reading Cocoacasts for a while, then you know that I'm not a raving fan of this design pattern. It falls short in a number of ways, and I mostly use the **Model-View-ViewModel** pattern instead.

The Model-View-Controller pattern is easy to pick up, and it's a good fit for building Cocoa applications. Apple's SDKs are impregnated with this pattern and most of us have used the Model-View-Controller pattern for years and years, and not only for building Cocoa applications. It's a widely used, versatile design pattern that has earned its stripes over several decades.

Delegation

Another very common pattern you encounter in Apple's frameworks and libraries is delegation. The concept is simple, and it promotes loose coupling and code reusability. If you've used table views in your application, then you're already familiar with delegation. A table view isn't responsible for responding to user interaction. It delegates this task to a delegate object. When a user taps a row in the table view, the table view

notifies its delegate about this event. It's up to the delegate object, a view controller, for example, to decide how the application should respond.

The data source pattern is similar. It's used to populate a table view with data. A table view is a pretty dumb object. That's a good thing because it promotes reusability. It asks its data source what data it needs to present to the user. The table view only knows how to present the data it's given by its data source.

Notifications

While delegation is a good fit for objects that have a one-to-one relationship, notifications are ideal for distributing messages to one or more unrelated objects. Know and understand what the differences are. Delegation and notifications aren't interchangeable design patterns.

Like delegation, notifications are very common in Apple's frameworks and libraries. They're an essential ingredient of every Cocoa application.

KVC and KVO

Key-value coding and key-value observing are related concepts that power the Cocoa frameworks. Unfortunately, they're almost always overlooked by developers new to Cocoa. For that reason, I spend some extra time discussing these concepts later in the book. Make sure you don't skip KVC and KVO if you're not quite sure what they are.

Xcode

One of the first tasks you need to take care of if you decide to build a Cocoa application is downloading, installing, and launching Xcode. Developers rarely take the time to learn the ins and outs of Apple's powerful IDE (Integrated Development Environment). Spend some time exploring the basics of Xcode. There are many, many tips, tricks, and features you only pick up after using Xcode for a while, but it's nevertheless important to learn the basics.

There Is No Clear Path

We live in times where we're bombarded with information. If you're learning a new technology, framework, or tool, you only need to enter a few words into Google, and you're presented with dozens if not hundreds

of resources, many of them free. The problem is that most of us struggle with this abundance of information. Which resource should we choose? How do we separate the good from the bad? Should I learn something else first?

The mobile space is evolving at such a rapid pace that many of us are scrambling to keep up. To be honest, I don't feel that. The reason is simple. I invest in my foundation, and that continues to pay dividends. Learning a new framework with a solid foundation to rely on is much easier. I can't repeat this enough, invest in your foundation. Every. Single. Day.

Once you have that foundation, everything automatically becomes easier. You'll notice that there's nothing you cannot master. Apple's documentation no longer looks mysterious. It starts to make sense.

Focus

When Apple released the iPhone in 2007, we didn't have an SDK to work with. The more experienced developers among us reverse engineered the operating system and built applications for the first iPhone. Ten years later, Apple has given developers access to a rich ecosystem with the tools we need to build amazing software. As a Cocoa developer, you can build software for your computer, your phone, your tablet, your watch, and even for your television. These are amazing times. But, again, it's easy to be overwhelmed. Focus is important.

I know several developers that build software for both Apple's and Google's ecosystems. While this may seem appealing, it isn't something I recommend. It's very challenging to stay up to date about one platform let alone two. There certainly are advantages to this approach, but it's more important than ever to specialize.

Your attention is a valuable asset, and I recommend to focus, focus aggressively. If your employer expects you to be up to date on iOS, tvOS, macOS, and watchOS, then they're expecting too much. Chances are that they're not technical or don't know much about Apple's ecosystem. Gone are the days that you know every framework and library inside out.

I choose to focus on Apple's ecosystem, with a strong emphasis on iOS. I also build tvOS and watchOS applications, and the occasional macOS application, but those aren't my main focus. This focus allows me to build robust, performant applications.

Apple applies that same focus, or it used to. Choosing a new Apple computer used to be a simple task because you only had a handful of options. While that's no longer true, it emphasizes the power of focus and simplicity. And it's the reason I choose to build a career on Apple's ecosystem.

Which Path to Follow

The most common question, or frustration, I hear from developers new to Cocoa and Swift is "Where should I start?" There's so much to learn, or that's how it seems. You have the language, the tools, the frameworks, the App Store. There doesn't appear to be a clear path; there's only a lot of information. Platforms like Stack Overflow are fantastic, but it's important to learn to digest end-to-end information.

> There doesn't seem to be a clear path, there's only a lot of information.

I talk more about this later in the book, but I need to repeat myself by saying that focus is essential. If you're starting out as a developer, then make sure you're not jumping from topic to topic. Choose a proven path. A resource I often recommend to developers is the book about Cocoa development published by Big Nerd Ranch. It's this book by Aaron Hillegass that taught me Cocoa development. Aaron and his team have decades of experience teaching people Cocoa development.

There are three reasons for choosing a trusted source. First, you learn the fundamentals. Second, you can ignore the abundance of information and focus. Third, you're sure you're learning things the right way. Big Nerd Ranch, for example, has been around for almost two decades, an eternity in the technology industry.

Focus, focus, focus. That's what's going to keep you sane in today's world. That's one of the first things I teach the students I coach. It's fine to

ignore most of what you read and focus on one thing at a time. Your productivity will skyrocket, and your confidence will follow suit.

2 Where to Start

In this chapter, I describe the path I recommend to people that are new to Cocoa and Swift development. If you're new to programming, then I recommend taking a basic course on programming first.

The path I present in this chapter requires you to learn a collection of skills, patterns, and paradigms. Remember from the previous chapter that it's essential that you remain focused. It's tempting to jump from topic to topic because there's a lot you need to learn.

Setting Yourself Up for Success

Before we explore the path to learn Cocoa and Swift development, it's important to understand that it takes time. Not only does it take time to learn the various topics I cover in this chapter, it takes time to let everything sink in.

You will encounter concepts that only click a day, a week, or a month after you've read about them in a book or seen them explained in a video. Don't let yourself become frustrated if you don't get everything right away. That's perfectly normal.

> Don't let yourself become frustrated if you don't get everything right away. That's perfectly normal.

The Swift language is easy to pick up, but the finer details and more advanced features are more challenging. The Cocoa SDK with its many frameworks and APIs can seem daunting, but if you know what to focus on when and understand why you're learning something, then the pieces of the puzzle start to fit together, showing you the bigger picture.

A Path Without Distractions

The path I almost always recommend to students new to Cocoa and Swift development shows you how to build an application from start to

finish. I strongly believe that every developer needs to become familiar with every step of building and deploying an application.

If you decide to work at a larger company as an employee, you probably won't have to deal with every step of this process. However, knowing about and understanding the steps involved to build an application is an integral part of your foundation. It's essential that you understand every step of the process, and there's no better way to learn the steps than to go through the process yourself.

Step 1: Learn the Language

The first step is the most obvious one, learning the Swift language. Learning the language and the frameworks at the same time isn't something I recommend, unless you're already familiar with Objective-C. That's why I always recommend to learn the basics of the Swift language first. The Swift Programming Language is a very good starting point and it's what I recommend you read. If you're new to Swift and Objective-C, I recommend skipping **A Swift Tour** because it will confuse you more than it will teach you.

The Swift Programming Language is available from the Swift website and you can download it for free from iTunes. But don't take the book and read it from cover to cover. Take out your computer, install Xcode, and write code. Let everything sink in by applying what you learn. You only learn the language by writing it, using its syntax, and becoming familiar with the various concepts. Write, write, write.

This also means it's time to install Xcode. Visit the Apple Developer website and download a copy of Xcode. You can also download Xcode from the Mac App Store. Fire up Xcode, create a playground, and write code as you read **The Swift Programming Language**. This is one of the best approaches to learn the language.

I browse **The Swift Programming Language** several times a week. Because it contains so much information, it's virtually impossible to learn and understand everything about the language after one read.

When is it time to continue to the next step? You're ready to move on if you understand the basics of the language. How do you know if you do?

It means that you understand and can apply the following concepts and paradigms.

- You understand the difference between variables and constants, and you know when it's appropriate to use which.
- You know what optionals are, why they're integral to the language, and you also understand what the exclamation mark stands for. You also need to understand optional chaining, safely unwrapping, and the meaning of `nil`.
- You understand and can use strings, arrays, dictionaries, and sets. You also know about object literals and how to create them.
- You can use control flow to add complexity to the code you write.
- You understand the basics of functions and closures, and you can use them.
- You know what classes, structures, and enumerations are and what sets them apart.
- You understand what reference types and value types are, you know the pros and cons of each, and you get what sets them apart.
- You understand the basics of protocols. You can define a protocol and create a type that conforms to that protocol.
- You also know about extensions and how to use them.

You'll learn a lot more along the way, such as error handling, inheritance, properties and methods. You should now be ready to build your first application.

Step 2: Build an Application

Once you have a basic understanding of the Swift language, it's time to start building something that runs on an iPhone or an iPad. Even though playgrounds are fantastic to learn Swift, you started this journey to build applications. One of the most exciting moments of my journey as a software developer was the day I ran my first application on a physical device.

I encourage you to create your first application as soon as you have a good grasp of the language. The application won't do much, but it puts the language and the frameworks together. It should also get you excited

about what's to come. There's nothing as exciting as running your very first application on your phone or tablet.

Step 3: Explore the SDK

The application you build in the previous step won't make a dent in the universe, but it whets your appetite. It's time to become familiar with the frameworks and tools you need to build robust applications, which includes learning about the **Foundation** framework, the Swift **standard library**, and **UIKit** (iOS) or **AppKit** (macOS). A good book or course on Cocoa development covers the important aspects of these frameworks and tools.

For iOS developers, it's essential to be familiar with views and view controllers. Understand how a view relates to a view controller and how both fit into the Model-View-Controller pattern. While you could create the user interface in code, I don't recommend this. Apple pushes developers to storyboards and that's something you need to become familiar with. But don't ignore XIB files. They're a fine alternative if you don't like working with storyboards.

Auto Layout is a layout engine that makes building user interfaces easier. You need to become familiar with constraints, layout guides, and view hierarchies. Don't skip this step. Many developers do and pay the price later.

Almost every iOS application includes a table view or a collection view. As you learn how they work, you also pick up the delegation and data source patterns, which we discussed in the previous chapter.

Navigation controllers are another essential ingredient of most iOS applications. It may be challenging to wrap your head around the concept, but once it clicks you'll find them easy to use. The same applies to tab bar controllers.

Storing data is a task of most applications. There are many solutions available. The simplest solution provided by the Cocoa SDK is the defaults system. The defaults system is nothing more than a key-value store. You'll find yourself using it very often. I also recommend learning

about more advanced types of data persistence, such as storing data in a file on disk or in a database.

Later in this chapter, I emphasize how important it is to choose a resource that teaches you these fundamentals. It pays to invest in a good book or course. You're investing in your foundation.

Step 4: More Challenges

After you understand the fundamentals covered in the previous step, it's time for a slight detour to learn about a few more advanced, yet essential, aspects, including memory management, asynchronous programming, and data persistence. These are more advanced topics, but you need to learn them as they're an ingredient of every application that has a bit of complexity to it.

I repeat, pick a good book or course to learn these concepts. It'll make your journey that much easier. Memory management and multithreading are not easy to grasp for an inexperienced developer. But believe me when I say that you can do this.

Step 5: Build That Application

It's time to accelerate your learning by building an application. This step is the cherry on the cake. You start with an empty Xcode template and start translating your idea to a functional product. It doesn't matter what idea you have but it needs to be challenging.

It doesn't need to be unique, though. A good example is an application to take notes or view the weather. The goal is to apply what you have learned in the previous steps.

Be prepared. You will run into problems and you will need to learn some more to finish the project. It will take you days or weeks to complete this step. I hope this doesn't scare you because that's the reality of software development.

If you're a developer, then you'll need to continue learning on a daily basis. This is inevitable, especially in a space that moves a breakneck speed.

Step 6: What About Testing

Testing is often skipped for some reason. If testing were a difficult subject, I'd understand why that is. But it isn't. I feel it's usually skipped because it isn't a cool topic. Don't skip this step. Testing isn't as hard as you think. Later in this book, we cover testing in a bit more detail.

Step 7: Publish

If your application is ready to be shared with the world, it's time to publish it on Apple's App Store. This is quite challenging because you need to use a bunch of tools to publish your application. You need to visit the Developer Portal, provision your application, and prepare your application in iTunes Connect. This isn't rocket science, but it can be frustrating the first time you try to use these tools. Don't worry, though. You can do this too.

Choose Your Teacher Wisely

While you can go through these steps on your own, I recommend that you look for help. You can learn the Swift language by reading **The Swift Programming Language**. Learning the Cocoa SDK is a bit more challenging. That's the step in which most developers lose track.

Pick up a book or course that covers the fundamentals in such a way that you learn everything you need to build a simple yet functional application. Later in this book I talk about the importance of choosing your teacher. Make sure you learn from someone you trust. That someone teaches you best practices and they also show you which practices to avoid. Your teacher keeps you sane and confident as you learn your craft.

I can recommend anything published by Big Nerd Ranch, such as iOS Programming: The Big Nerd Ranch Guide. That said, there are various books and courses that are very good. Don't pick any book or course. Make sure you team up with someone who knows his or her stuff.

Don't pick or choose random tutorials you find on the web. I strongly recommend choosing for a solution that show you the complete path. The result is almost always confusion and frustration if you pick and choose.

This approach is fine once you've nailed the fundamentals. As a beginner, though, I recommend sticking with a proven solution.

If you have money to spend and you want to speed up your learning, you can enroll in a bootcamp to immerse yourself in Swift development. The result is that you learn a lot in a very short period.

3 Know Your Tools

An important part of your foundation is knowing the tools you work with. And this goes beyond just Xcode. I always encourage developers, regardless of the company you work for, to know about every aspect of building, testing, and shipping a Cocoa application. This chapter covers the tools you need to know about and become familiar with. Let's start with the workhorse of every Cocoa developer, Xcode.

Xcode

Xcode is the most popular IDE for building Cocoa applications. There are alternatives, such as JetBrains' AppCode, but I won't cover those in this book. While Xcode has served me well over the years, there is, and has always been, room for improvement.

While it's important that you become familiar with Xcode's user interface, it's not important to spend hours or days learning the ins and outs of Xcode. My experience has taught me that developers usually don't have a hard time getting up to speed with Xcode. The basics are the same for most IDEs and you learn about the more intricate features of Xcode over time.

But that doesn't mean you shouldn't dig a bit deeper from time to time. Xcode has a slew of hidden features and keyboard shortcuts. It's worth spending an hour here and there reading up on Xcode, watching a WWDC session, or simply playing around with Xcode's user interface.

I'd like to highlight a few components of Xcode that deserve special attention and that you need to know about.

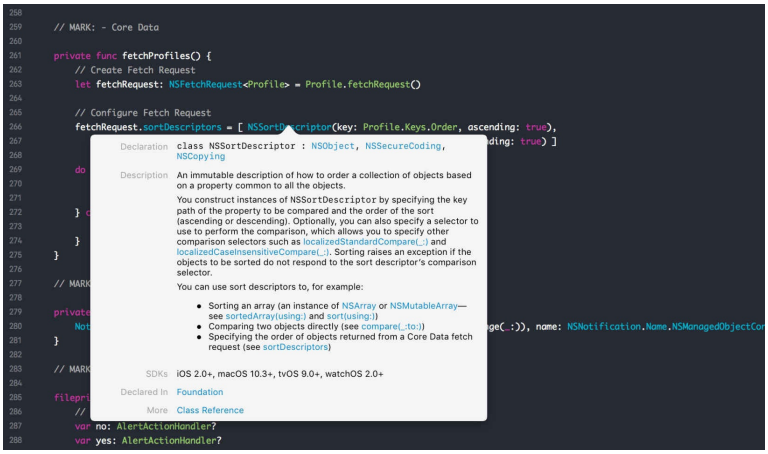
Documentation

Documentation is a developer's bread and butter. Xcode's built-in documentation browser hasn't always been great. Fortunately, Xcode's documentation browser received an overhaul in Xcode 8 and 9. I used to

use Dash to explore the Cocoa SDK, but I've switched to the built-in documentation browser since Xcode 8. It's nicely structured, fast, easy to use, and it supports tabs.

You can bring up the documentation browser from within the source editor by pressing **Option** and clicking a symbol. This shows you a summary of the symbol. You can open the documentation browser by clicking one of the links in the pop-up window.

Xcode's Documentation Browser



Xcode's Documentation Browser

You can also open the documentation browser by pressing **Shift + Command + 0** or choosing **Developer Documentation** from Xcode's **Window** menu.

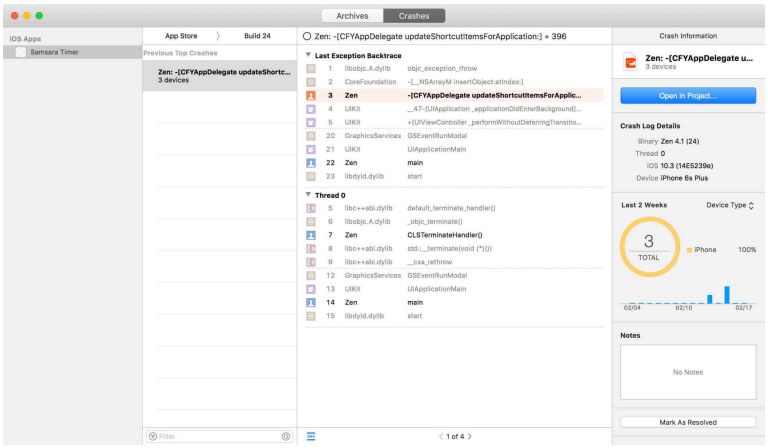
Devices

The **Window** menu also gives you access to the **Devices and Simulators** window. This is useful if you need to download the container of an application or add a simulator.

Organizer

Xcode's **Organizer** is more powerful than many developers know or think. It shows you a list of archives, but it also provides easy access to

crash reports. The crash reports neatly tie into Xcode's source editor, which is very convenient for debugging fatal issues.



Xcode's Organizer

The organizer also allows you to export your application or send it to iTunes Connect for testing or deployment to Apple's App Store.

Developer Portal

Even though Xcode can communicate with Apple's servers on your behalf, you sometimes need to visit Apple's developer portal to take care of specific tasks, such as creating a certificate or configuring an App ID.

It used to be tedious to work with the Developer Portal. This has become easier and Xcode takes care of many trivial tasks, such as adding capabilities to an App ID and generating provisioning profiles.

Provisioning

Most developers cringe when someone drops the word *provisioning*. An application needs to be provisioned for it to run on a device. This is a necessary evil if you're developing for Apple's ecosystem. It's tedious, it drives you mad from time to time, but it's important that you understand why it's necessary, how it works, and how you can resolve common issues.

This is one of those topics that's usually skipped or ignored, that is, until things hit the fan. You don't want to learn about provisioning the moment you need to deploy a hotfix to the App Store. If you have some downtime,

spend an hour reading about and learning the ins and outs of application provisioning. You'll be thankful that you did when things go haywire.

Instruments

Xcode ships with several other developers tools, including Instruments. Instruments is one of the most underused and undervalued tools in a Cocoa developer's toolbox. It's powerful, versatile, and a great help for debugging more complex issues.

While Instruments is useful, I wouldn't focus on this tool from day one. In fact, I wouldn't invest much time in it in your first year unless you need it for debugging. There are more important things to learn in the first year of your journey. Once you're familiar with Xcode and Cocoa development, though, it's certainly worth investing time in exploring Instruments.

Command Line

While it's possible to create Cocoa applications without knowing a thing about the command line, I've always found it a big asset to be familiar with, and not afraid of, the command line. Interacting with CocoaPods or Carthage, for example, is mostly done from the command line. Some developers swear by the command line for many other tasks, such as source control and running tests.

This isn't something you need to invest time in right now, but have an open mind about the use of the command line. If you have some time to spare, then explore a few tutorials about the command line and how you can use it to your advantage.

Third Party Tools

Even though Xcode works fine for Cocoa development, there are several tools that make building Cocoa applications easier and more enjoyable. I don't have any affiliation with the companies that create these products.

Tower

Tower is my favorite Git client and it gets better with every release. I use the command line for simple Git operations, but most of the time I use Tower to get the job done. It's a perfect fit for my workflow.

Reveal

You probably know that Xcode includes a view debugger, allowing you to inspect the view hierarchy of a Cocoa application. Reveal is a standalone application that performs a similar task, but it does this much, much better. Reveal has been around for several years and it also includes support for tvOS applications.

Charles

Charles is a cross-platform application for inspecting network traffic. It isn't always easy to get it up and running with a physical device, but, once set up, it works very well and can be a lifesaver for debugging network issues.

PaintCode

Writing drawing code is tedious and verbose. PaintCode makes this a breeze. PaintCode is a drawing application that converts your artwork into drawing code for several platforms. It supports a range of languages, including Swift and Objective-C.

4 Adopt Best Practices

Whenever someone signs up for Cocoacasts, I ask them what they'd like to achieve in the next three, six, or twelve months. Many developers tell me they want to become a better Swift developer. This is a vague ambition and I usually ask them what that means for them. What does it mean to become a better Swift developer?

A common answer is that they want to learn about patterns and best practices. That's an interesting answer because it shows that they have the ambition to grow as a developer and they know that their current code can use some improvements.

What Are Best Practices

The term *best practices* is overused in my opinion. I prefer to use the term *good practices*. What I like about software development is that there's rarely one solution to solve a problem. That also implies that there are no *best* practices, only *good* practices and habits.

Apple unveiled Swift only three years ago. Unless you're Chris Lattner, the language is still very new to us. Many of us are still looking for good practices to adopt. But how do you spot a good practice? Some developers intuitively spot a good practice when they see one. It's often a solution to a problem that feels right. It looks elegant and, more often than not, it's easy to implement.

Take dependency injection as an example. This is one of the patterns I teach in my book Learn the Four Swift Patterns I Swear By. Even though many developers have a hard time wrapping their head around the concept, dependency injection is easy to adopt and implement. It's a cure for tight coupling and an effective solution to rid a project of singletons.

Good practices aren't necessarily the easiest or most convenient solutions you find. Have you heard about the singleton pattern? This pattern is easy to adopt and it's very convenient to use in a project. But

there’s a caveat. Convenience is sometimes a code smell. I mention this several times in the book. If something’s convenient, then you may need to step back and take a hard look.

You shouldn’t avoid convenience, but you need to be critical about the patterns you use and the practices you adopt. Even if you have little experience, you need to scrutinize what you pick up. It usually only takes a healthy dose of common sense to distinguish a pattern from an anti-pattern. Be critical and be honest with yourself.

Patterns

Patterns are an important aspect of software development. They’re proven recipes to solve problems. They allow you to get work done instead of reinventing the wheel over and over.

Later in the book, I zoom in on several patterns I strongly recommend you take a look at. They have become an integral part of my development workflow. It doesn’t mean that you need to use or adopt these patterns in your projects, but it can be eye-opening to explore them and learn from them.

Anti-patterns

There are very few patterns I recommend that you avoid altogether. They’re often referred to as anti-patterns. There’s one pattern that I have a strong opinion about, the singleton pattern. While I don’t consider the singleton pattern a bad pattern or an anti-pattern, I advise to use it sparingly. Very, very sparingly.

The primary reason the singleton pattern has a bad reputation is misuse. A common reason developers adopt the singleton pattern is convenience. Do you remember what I said about convenience?

You probably know that the use of globals is an anti-pattern in most programming languages. The singleton pattern gives you global access to the singleton object. Doesn’t this start to smell? Unfortunately, this side effect of the singleton pattern is the primary reason many developers adopt the pattern. Don’t do that.

Good Practices for Swift

As I mentioned earlier, with the release of Swift, many of us started looking for and reading about good practices. They were difficult to find at first, but, as the language evolved, it's clear what you should and shouldn't do.

Embrace Optionals

For some, Swift is almost synonymous to optionals. For others, optionals are synonymous to frustration. Optionals are an essential aspect of Swift's obsession with safety. And that's a good thing. The ability to send messages to `nil` in Objective-C is convenient, but it's heavily misused. It's very often an easy way out for developers.

Swift puts a stop to this. The language is very clear about the meaning of `nil`, the absence of a value, and it's up to you to deal with it. Plain and simple.

Developers new to Swift often take refuge to the exclamation mark. This isn't something I recommend. In fact, as I mention elsewhere in the book, I see most uses of the exclamation as an anti-pattern. The exclamation mark is a necessary element of the Swift language, but, as with every invention, it's often misused.

Learn the Language

The most common reason developers struggle with Swift is because they don't know the language yet. There's nothing wrong with that since many of us are still learning. However, whenever I find myself cursing the compiler for yet another mysterious error message, I look for a solution. The language is still young, but the early days of the language, when it was barely usable, are over.

Swift 3 and 4 are a joy to use and I don't very often find myself struggling with the language itself. I browse the language guide or look for a solution on the web. There's almost certainly a solution that fits your current need.

That's also why I recommend investing in your education. I know that work can get in the way from time to time, but I strongly recommend allocating a chunk of time every day or every week dedicated to learning something new. And I don't mean learning a new shiny framework, such as ARKit. Invest in your foundation. That doesn't sound as fancy as ARKit, but it makes your job more fun in many subtle ways.

Swift's Core Ingredients

The more I use Swift, the more I fall in love with the language. Value types are one of the aspects I adore about Swift. We have structs and enums in Objective-C, but they're not nearly as powerful or easy to use as their counterparts in Swift. Enums are fantastic and structs are a good fit for a wide range of problems.

Value types and reference types each have their place and we need references types to build Cocoa applications. The next time you define a class, though, consider creating a struct or enum. Would that also work?

Later in the book, I go into more detail about the differences and the benefits of value types and reference types. Use value types whenever you can and use reference types when it feels you need them. You don't need to choose one or the other. They complement one another quite well.

The Swift language is composed of a small number of core values or core ingredients. Once you discover what they are, you better understand how the language works best and should be used. I already mentioned value types and optionals. Later in the book, I also talk about protocols and protocol-oriented programming.

Be Critical

Being critical is a skill every developer needs to cultivate. Learn from others and experiment, but remain critical about the lessons you take away and adopt. I cannot emphasize this enough.

5 Security

Security is a fundamental aspect of software development and it's important to know about best practices and common patterns that can help strengthen the security of the projects you work on. I want to emphasize that I'm not a security expert. The recommendations I provide in this chapter are based on my experience and what I've learned from fellow developers.

Make It Hard

I once read that, if someone wants to access your data, then they will succeed. How badly they want to access your data determines whether they'll succeed. I don't know whether this is true, but I tend to err on the side of safety.

Why is this important? It changed my perspective on security. It's naive to think that you can outsmart people that are trained to find and extract the information they need. That doesn't mean you need to be complacent or ignore the advice you read. It simply means that your actions and motivation change slightly.

An effective approach to security is to have the mindset to make it hard for the other party to access the data you're trying to protect. In other words, you add several layers of security to protect the data of the user. Let's start with the basics.

Plain Text

If you have some experience developing software, you most likely know that you shouldn't store sensitive information in plain text. Ever. Don't store the user's username and password in the user defaults database, for example. Use the keychain to protect this type of sensitive information.

The same applies to networking. Apple and Google are actively forcing developers to move away from HTTP and use SSL by default. Apple's **App Transport Security** encourages developers to be aware of the security risks of their applications. Make sure that your application communicates with remote services over a secure connection. This isn't always possible if you aren't in control of the remote service. In such a scenario, it's up to you to decide what the next best option is.

But SSL may not always be sufficient. Your application is still susceptible to, for example, man-in-the-middle attacks. You can remedy this by adopting certificate pinning, adding an extra layer of security.

Obfuscating Information

A common question I receive is how to best hide or obfuscate sensitive information that's bundled with your application. That's a good question. The answer may disappoint you, though. As I mentioned earlier in this chapter, there's always a way for people with bad intentions to get a hold of the information they need. You need to consider the sensitivity of the information you're trying to protect.

The same advice applies, though. Make it as hard as possible. But, at the same time, consider the sensitivity of the information you're protecting. Don't store sensitive information, such as API keys, in your application's **Info.plist**. It's easy to dissect an application you downloaded from the App Store and inspect the contents of the **Info.plist**.

I usually store sensitive information as private constants in a configuration file, which means it's compiled alongside the application. This doesn't make it impossible to extract the sensitive information, but it makes it less trivial.

Fetching Sensitive Information

You can go one step further and avoid storing keys or credentials in the application itself. Instead, your application contacts a remote service and asks for credentials every time it needs to communicate with that service. This requires a dedicated infrastructure and a lot more work up front, but it adds a powerful layer of security.

Encryption

Encryption is an effective solution to protect the user's data. Realm, for example, has built-in support for encrypting the data stored in its database. For Core Data, however, this is less trivial. I hope Apple will make this less cumbersome in a future release of the framework.

The data the user stores on their device is automatically encrypted if the device is protected with a password or Touch ID. Only you, the user, can unlock the data stored on your device because you hold the key to decrypt it, not Apple. It's great to see that Apple continues to invest in the privacy and security of its customers. Apple's motivation is a bit more nuanced, though.

Privacy

A lot has been written about privacy and protecting the user's privacy. Unfortunately, many developers don't realize that this also means protecting the user's privacy from companies that offer services they use day in day out. If your application uses analytics or displays ads, then you're exposing the user's personal information to the companies behind these services.

I used to use Fabric for crash reporting and analytics, but I no longer do for personal projects. As a developer, it's my responsibility to protect the user's privacy and they expect that from me. I understand that many developers don't have this luxury, but I still believe that you should, at a minimum, consider the option and be aware of the information you may be exposing to third parties.

If you include a third party SDK in your application and you don't have access to the source, then how do you know what information you're sharing with this third party? You don't. That's important to keep in mind.

Logging and Debugging

Logging information to the console is my favorite technique to debug issues because it's simple and to the point. It's a technique many of us use, but it's also a potential security problem. Many developers forget that print or log statements also log information to the console in

production. This can be useful and intentional, but it can also be a security issue.

I hope you're not logging credentials or other sensitive information. Even fragments of the user's data shouldn't be logged in production. If you need to generate logs, then I recommend looking into remote logging in combination with data encryption. Avoid that a third party, any third party, can access the logs you generate.

Educating Your Client

The role of a developer is often reduced to writing code and solving a problem. Not only is this incorrect, I strongly believe any developer, regardless of their experience, should also provide a technical service to the parties they work with. What does that mean? If you're told to implement a solution, then it's your responsibility to inform your client or project manager about any security risks or problems.

I believe it's the task of the developer to educate the client. The client still decides what happens and what needs to be implemented, but they should at a minimum be aware of the risks involved. I've implemented several solutions I didn't agree with, but I tried to educate the client about alternative solutions that were safer.

At one point I inherited a project in which the user's credentials were stored in the user defaults database. Even though there was no room to refactor this glaring security hole, I informed the client about the problem. For a developer, it can be frustrating not having final say in such arguments, but that's how it is. This is very different if you build a product business in which you make the calls.

6 Don’t Ignore Your Foundation

It’s common for companies to build an in-house framework that lies on top of Apple’s frameworks. Some frameworks fix bugs and add conveniences to Apple’s APIs while others abstract away large portions of the native frameworks.

It’s clear why this is a common practice in large companies, especially agencies. You don’t want to reinvent the wheel every time you need, well, a wheel. But there are inherent risk involved that are often ignored or not spoken about.

Under the Hood

One of the most important pieces of advice in this book is simple but so important, invest in your foundation. Once you have a solid foundation, you’ll notice that picking up a new framework or solving a challenging problem is possible. It’s no longer complex, it’s simply challenging.

But if you land a job at a company that kindly forces you to use the in-house framework then your foundation is compromised. I’m sure you can build beautiful applications that work very well. But I wonder if you know *and* understand what’s happening under the hood? What happens if you move to a new company? An in-house framework is almost always the company’s intellectual property. Do you know how to build applications without it?

Dependencies

This brings us to dependencies, a topic I cover in more detail later in the book. I always try to minimize the number of dependencies I include in a project. Every dependency needs to earn its way onto the project. The reality is that most don’t make it into the project’s **Podfile** or **Cartfile**, which is a good thing.

I’m a big fan of lightweight abstractions or wrappers around frameworks. Core Data is a fine example. I don’t use a library to work with Core Data, but I create a handful of files that include convenience methods to interact with the framework. These are usually specific to the project.

Don’t Ignore the Fundamentals

Investing in the fundamentals is the best investment you can make as a developer, regardless of your experience. Even if the company you work for uses an in-house framework, I encourage you to continue investing in the fundamentals of the platform you build software for.

7 Respect the SDK

When you're new to a language or framework, it occasionally happens that you find yourself fighting the language or framework. Whenever you feel that you're coding yourself in a corner, you should stop and take a step back.

Take the Cocoa SDK as an example. The engineers at Apple aren't stupid and they carefully craft the APIs of the various libraries and frameworks. If you find yourself jumping through a lot of hoops to accomplish a simple task, then it's very likely that there's a better solution you're not aware of.

Know Your Limitations

At the same time, you should be aware of the limitations of the API you're using. This is difficult when you're starting out. Take the user defaults system as an example. The user defaults database is meant for storing small chunks of data. These are stored as key-value pairs.

While you can store complex object graphs in the user defaults database (as long as you stick to the supported types) don't be naive and store megabytes of data in the user defaults database. It'll probably work, but it won't be a performant or scalable solution.

Respect the SDK

A very common type of question on Stack Overflow is "How do I customize …?" Fill in the blank. Most of the components of the UIKit framework are very customizable, but there are times when you run into limitations. Apple does its best to keep the user interface of applications on its platforms consistent and by limiting the customization of user interface components it tries to discourage too much customization.

A few weeks ago, I was looking for a solution to a trivial problem. I wanted to modify the color of the clear button of a text field, an instance

of the `UITextField` class. Several Stack Overflow entries suggested to look for the clear button in the view hierarchy and setting its tint color property to the desired color. This is a bad idea and a common example of running into the limitation of an API.

The `UITextField` class currently doesn't support modifying the tint color of the clear button. That's a fact. The only option I had was filing a bug and move on. I ended up implementing a custom solution that solved the problem.

I'm still not sure if the inability to customize the tint color of the clear button is a deliberate choice made by Apple, but the truth is that the API of the `UITextField` class doesn't support it. That means that I'm not able to use the default implementation of `UITextField` if I need to customize the clear button. You decide if you want to push the limits, but you do so at your own risk.

Digging into the view hierarchy of the text field may resolve the issue today, and it did for many developers judged by the number of upvotes, but you're building a house on quicksand. The moment Apple makes changes to the internals of the `UITextField` class, your solution stops working. If you're lucky, the user will see the default clear button. If you're not so lucky, your application crashes or is rejected by Apple.

9 Design Patterns

In this chapter, I discuss some of the design patterns I use day in day out. They're my bread and butter. If you're a Cocoacasts reader or student, then you may already be familiar with some of them.

Model-View-Controller

I bet you're already familiar with **Model-View-Controller**, or **MVC** for short. The Model-View-Controller pattern is a widely used design pattern for architecting software applications. It's been around for several decades and made its way to many languages, frameworks, and libraries.

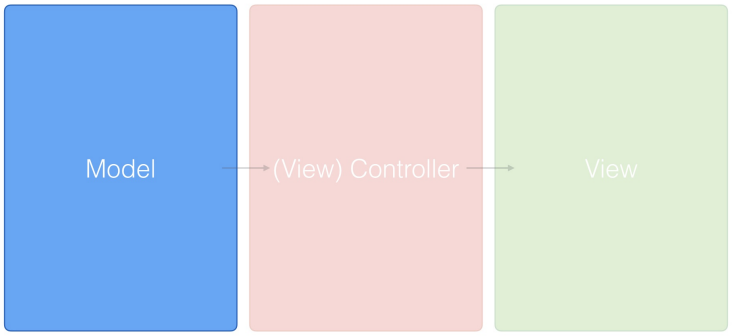
Cocoa applications are centered around the Model-View-Controller pattern, and many of Apple's frameworks make heavy use of MVC. It's therefore important that you know what it is, why it's a popular choice to architect software applications, and, let's be honest, where it falls short.

What Is MVC

As the name suggests, the Model-View-Controller pattern breaks an application up into three components or layers, **M**odel, **V**iew, and **C**ontroller.

Model

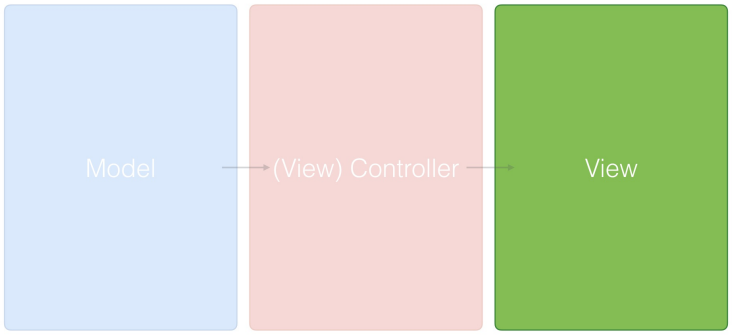
The model layer is responsible for the business logic of the application. It manages the application state, which also includes reading and writing data, persisting application state, and it may even include tasks related to data management, such as networking and data validation.



The M In MVC

View

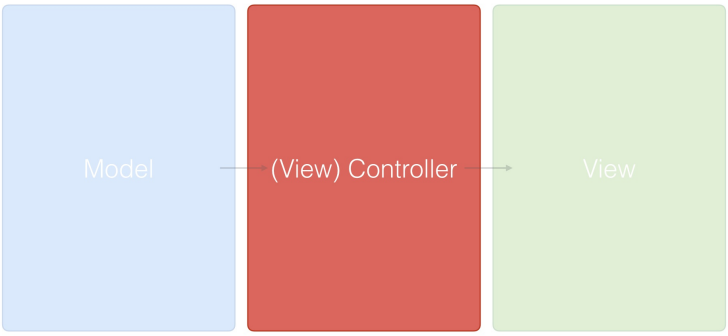
The view layer has two important tasks, presenting data to the user and handling user interaction. A core principle of the MVC pattern is the view layer's ignorance with respect to the model layer. Views are dumb objects. They only know how to present data to the user and they don't know or understand *what* they're presenting. This makes them flexible and easy to reuse.



The V In MVC

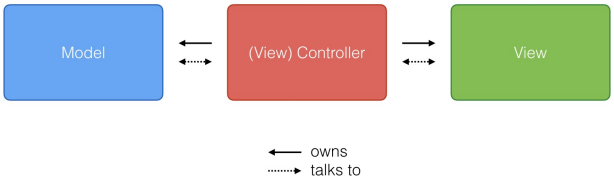
Controller

The view layer and the model layer are glued together by one or more controllers. In an iOS application, that glue is a view controller, an instance of the `UIViewController` class or a subclass thereof. In a macOS application, that glue is a window controller, an instance of the `NSWindowController` class or a subclass thereof.



The C In MVC

A controller knows about the view layer as well as the model layer. This often results in tight coupling, making controllers the least reusable components of an application based on the Model-View-Controller pattern. The view and model layers don't know about the controller. The controller owns the views and the models it interacts with.



Model-View-Controller in a Nutshell

Advantages

The Model-View-Controller pattern has earned its stripes over several decades and there are several reasons why it's such a popular design pattern.

Separation of Concerns

The most obvious advantage of the Model-View-Controller pattern is a clear separation of concerns. Each layer of the Model-View-Controller pattern is responsible for a clearly defined aspect of the application. In

most applications, there's no confusion about what belongs in the view layer and what belongs in the model layer.

What goes into controllers is often less clear. The result is that controllers are frequently used for everything that doesn't clearly belong to the view layer or the model layer.

Reusability

While controllers are often not reusable, view and model objects are mostly easy to reuse. If the Model-View-Controller pattern is correctly implemented, the view layer and the model layer should be composed of reusable components.

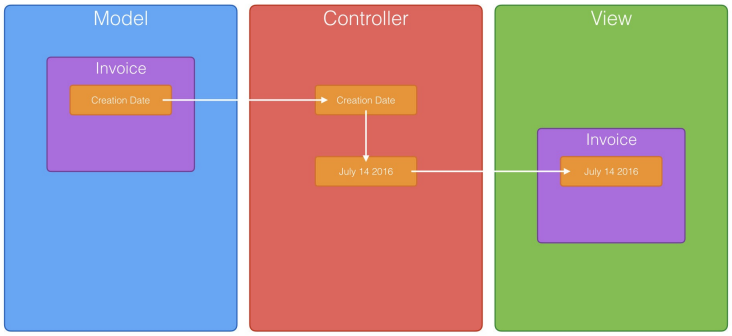
Problems

If you've spent any amount of time reading books or tutorials about iOS or macOS development, then you've probably come across people complaining about the Model-View-Controller pattern. Why is that? What's wrong with the Model-View-Controller pattern?

A clear separation of concerns is great. It makes your life as a developer easier. Projects are easier to architect and structure. That's only part of the story, though. Much of the code you write doesn't belong to the view layer or the model layer. No problem. Dump it in the controller. Problem solved? Not really.

An Example

Data formatting is a common task. Imagine that you're developing an invoicing application. Each invoice has a creation date. Depending on the locale of the user, the date of an invoice needs to be formatted differently.



An Example

The creation date of an invoice is stored in the model layer and the view displays the formatted date. That's obvious. But who's responsible for formatting the date? The model? Maybe. The view? Remember that the view shouldn't need to understand what it's presenting to the user. But why should the model be responsible for a task related to the user interface?

Wait a minute. What about our good old controller? Sure. Dump it in the controller. After thousands of lines of code, you end up with a bunch of overweight controllers, ready to burst and impossible to test. Isn't MVC the best thing ever?

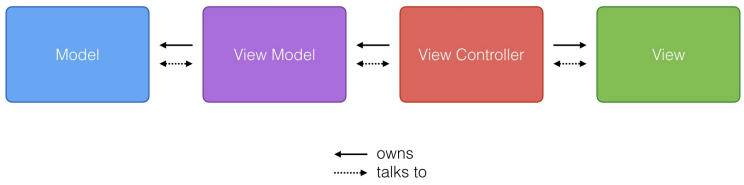
How Can We Solve This?

In recent years, another pattern has been gaining traction in the Cocoa community. It's commonly referred to as the Model-View-ViewModel pattern, MVVM for short. The origins of the MVVM pattern lead back to Microsoft's .NET framework and it continues to be used in modern Windows development. In the next section of this chapter, we take a closer look at the Model-View-ViewModel pattern and the advantages it has over the Model-View-Controller pattern.

Model-View-ViewModel

How does the Model-View-ViewModel pattern solve the problems we described earlier? The Model-View-ViewModel pattern introduces a fourth component, the **view model**. The view model is responsible for

managing the model and funneling the model's data to the view via the controller. This is what that looks like.



Model-View-ViewModel in a Nutshell

Despite its name, the MVVM pattern includes four components or layers, the **M**odel, the **V**iew, the **V**iew **M**odel, and the controller.

The implementation of a view model can sometimes be straightforward, translating data from the model to values the view layer can display. The controller is no longer responsible for this ungrateful task. Because view models have a close relationship with the models they consume, they're often considered more model than view.

Let's take a look at the advantages MVVM has over MVC. Why would you even consider trading the Model-View-Controller pattern for the Model-View-ViewModel pattern?

Advantages of MVVM

We already know that the Model-View-Controller pattern has a few flaws. With that in mind, what are the advantages MVVM has over MVC?

Better Separation of Concerns

Let me start by asking you a simple question. What do you do with code that doesn't fit or belong in the view or model layer? Do you put it in the controller layer? Don't feel guilty, though. This is what most developers do. The problem is that it inevitably leads to massive controllers that are difficult to test and manage.

The Model-View-ViewModel pattern presents a better separation of concerns by adding view models to the mix. The view model translates the data of the model layer into something the view layer can use. The controller layer is no longer responsible for this task.

Improved Testability

View (iOS) and window (macOS) controllers are notoriously hard to test because of their close relation to the view layer. By migrating some responsibilities, such as data manipulation, to the view model, testing becomes much easier. Testing view models is surprisingly easy. Testing? Easy? Absolutely.

Because a view model doesn't have a reference to the view controller that owns it, it's easy to write unit tests for a view model. Another benefit of MVVM is improved testability of view and window controllers. The controller no longer depends on the model layer, which makes them easier to test.

Transparent Communication

The responsibilities of the controller are reduced to controlling the interaction between the view and model layer. The view model provides a transparent interface to the view controller, which it uses to populate the view layer and interact with the model layer. This results in a transparent communication between the four components or layers of your application.

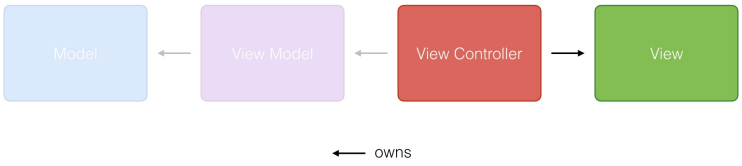
Basic Rules

There are six key elements that define the Model-View-ViewModel pattern. I sometimes refer to these as *rules*. However, once you understand how the Model-View-ViewModel pattern does its magic, it's fine to bend or break some of these rules.

Rule #1

First, the view doesn't know about the view controller it's owned by. Remember that views are supposed to be dumb. They only know how to present what they're given by the view controller to the user. This is a rule you shouldn't break. Ever.

#1

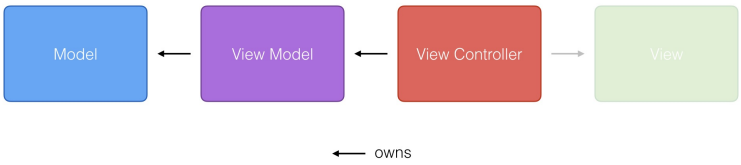


The view doesn’t know about the view controller it’s owned by.

Rule #2

Second, the view or window controller doesn’t know about the model. This is something that separates MVC from MVVM.

#2

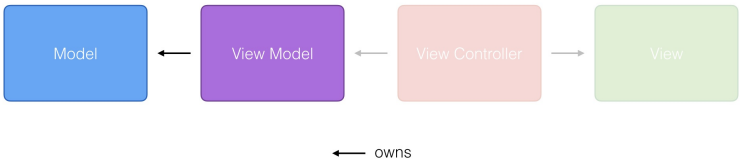


The view controller doesn’t know about the model.

Rule #3

The model doesn’t know about the view model it’s owned by. This is another rule that you shouldn’t break. The model should have no clue who it’s owned by.

#3

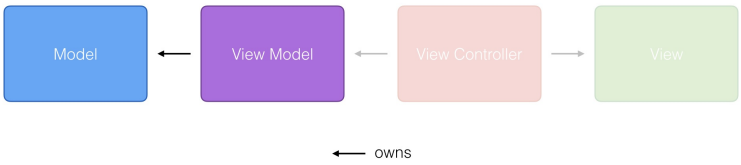


The model doesn’t know about the view model it’s owned by.

Rule #4

The view model owns the model. In a Model-View-Controller application, the model is usually owned by the view or window controller.

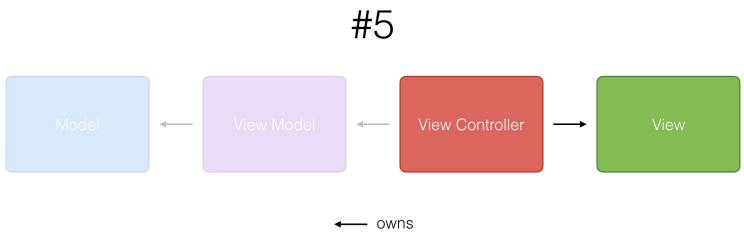
#4



The view model owns the model.

Rule #5

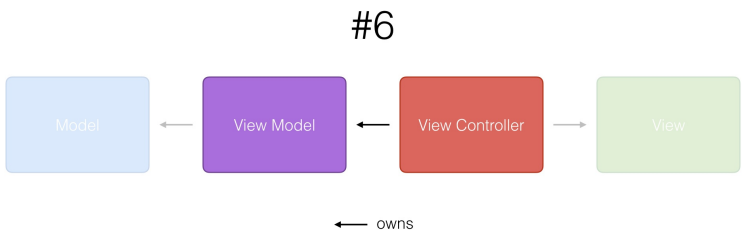
The view or window controller owns the view or window. This relationship remains unchanged.



The view controller owns the view.

Rule #6

And finally, the controller owns the view model. It interacts with the model layer through one or more view models.



The controller owns the view model.

Give It a Try

If you're new to Cocoa and Swift development, then I recommend sticking with the Model-View-Controller pattern. It's a proven pattern that many of us have used for years to build software. But if you notice that you're starting to hit the limits of the Model-View-Controller pattern, then it may be time to look for an alternative.

The Model-View-ViewModel pattern has changed how I write and think about software. Even though the changes it introduces are modest, the results are substantial. Combine the Model-View-ViewModel pattern with

reactive programming, which I discuss later in this chapter, and you have a potent recipe to build software.

Singletons

The singleton pattern is a pattern developers quickly become familiar with. When I first started dabbling with Cocoa development, I almost immediately came into contact with the singleton pattern. Many Cocoa frameworks, including UIKit and Foundation, use the singleton pattern. Because the Cocoa SDK makes use of the singleton pattern, developers think that it's fine to use singletons. It *is* fine to use singletons, but it's more nuanced than this.

What Is the Singleton Pattern

The concept is very simple. The singleton pattern ensures only one instance of a class is instantiated for the lifetime of the routine or application. The Cocoa frameworks often refer to a **shared object** or a **shared instance**. Take a look at these examples. `URLSession` and `UserDefaults` are both defined in the **Foundation** framework.

```
1 // Shared Session Object
2 let session = URLSession.shared
3
4 // Shared Defaults Object
5 let userDefaults = UserDefaults.standard
```

With the above definition in mind, the singleton pattern appears to be a useful design pattern. Unfortunately, many developers misuse the singleton pattern and use it to conveniently access an object from anywhere in a project. Having global access to a singleton is nothing but a **side effect** of the singleton pattern. It's not what the singleton pattern is about.

Singletons Everywhere

Whenever I talk about singletons, I like to bring up Maslow's hammer analogy.

> I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail. â€” Abraham Maslow

A surprising number of developers struggle with this problem. The singleton pattern appears to be solving a common problem, accessing an object in various places of a project. A singleton, nothing more than a fancy global, looks like the perfect fit for the problem. And it *is* the perfect fit if all you want to do is solve *that* problem. There's more to the story, though.

Have you ever wondered why some experienced developers consider the singleton pattern an anti-pattern? Why is it that such a useful pattern is often avoided by more senior developers?

Sacrificing Transparency for Convenience

By using singletons, you almost always sacrifice **transparency** for **convenience**. This is a problem I bring up frequently in this book. The question is, "How much are you willing to sacrifice for that little bit of convenience?" Convenience shouldn't rank high on your priority list if you're working on a software project. Let me show you with an example what the problem is.

> By using singletons, you almost always sacrifice transparency for convenience.

A common problem in software projects is user management. This means that a user model object needs to be created and managed. There are many solutions to this problem. If you're a fan of the singleton pattern, then you might create a singleton user object or a manager class that manages the currently signed in user. Only a *single* user can be signed in at a time. Right?

Problem solved? It's true that you can now access the currently signed in user from anywhere in your project. While this may seem like the best thing since sliced bread, it introduces several problems.

Consider the following question. Which object is allowed to modify the user model object? Easy. The manager object managing the user. Hmm … but isn't the manager object accessible from anywhere in your project? This means that the user model object can be modified from anywhere in your project. If you're still with me, then I hope you can see that this is

starting to sound less and less like the great idea it initially appeared to be.

How are you going to make sure that the objects that depend on the currently signed in user are notified when the user is modified? Easy. Notifications. Another popular option is to use a pattern similar to Java's observer pattern. You could use key-value observing, but do you really want to observe the properties of an object that's managed by a singleton? That sounds like asking for trouble.

Losing Track of Everything

I agree that the singleton pattern seems convenient and it may occasionally be warranted to use it, but the drawbacks almost always outweigh the benefits. Unfortunately, the drawbacks are very subtle at first and that's what misleads many developers.

> Unfortunately, the drawbacks are very subtle at first and that's what misleads many developers.

The most important drawback of the singleton pattern is sacrificing transparency for convenience. Consider the earlier example. Over time, you lose track of the objects that access the user model object and, more importantly, the objects that modify its properties.

The initial advantage of using a singleton, convenience, becomes the most important problem. Ironic. Isn't it. By using singletons in your project, you start to create technical debt. Singletons tend to spread like a virus because it's so easy to access them. It's difficult to keep track of where they're used and getting rid of a singleton can be a refactoring nightmare in large or complex projects.

Once the singleton pattern becomes an accepted practice in a project, it's difficult to move the project forward without using even more singletons. The structure of the project may seem flexible and loosely coupled, but it's anything but that.

Substituting Singletons With Dependency Injection

As I mentioned earlier, singletons have the tendency to spread like a virus. The cure to solve singleton disease is **dependency injection**. I cover dependency injection in more detail a bit later in this chapter.

While a key advantage of the singleton pattern is convenience, the most important advantage of dependency injection is **transparency**. I like transparency.

If an object requires a valid user model object to do its job, then that user model object should be injected as a dependency. It's easy to translate this requirement into code. Take a look at the following example.

```
class User {
    var firstName = ""
    var lastName = ""
}

class NetworkController {

    let user: User

    init(user: User) {
        self.user = user
    }

}
```

This snippet shows any developer working with the `NetworkController` class that it depends on a valid `User` instance. This is also known as **passing an object by reference**. It's less convenient than using a singleton, but it pays dividends in the long run. It adds clarity to the codebase, unambiguously showing which objects depend on which objects.

It's an advantage that passing an object by reference is less convenient. Huh? It forces you to consider your decision. Does the `NetworkController` class need access to the user model object? Would it suffice to pass the user's username and password instead? Dependency injection can be a useful tool to define the requirements of an object.

> Dependency injection can be a useful tool to define the requirements of an object.

Moving Away From Singletons

A few years ago, I started working on a client project that was littered with singletons. Some singletons even referenced other singletons, adding to the problem. After several rounds of refactoring, I managed to eliminate most of the singletons from the project, which drastically increased transparency and robustness.

Have you ever worked on a project nobody in your team wanted to touch? Modifying a handful of lines in one class could cause mayhem in a distant, unrelated component of the project? This is not uncommon for projects that make heavy use of singletons.

Another benefit of ridding a project of singletons relates to testing. Writing unit tests without having to worry about singletons is fantastic. It really is. Combine this with dependency injection and you have a great combination to work with.

Are Singletons Bad

My stance on singletons varies from day to day. If I had a rough day battling *singletonitis*, I dislike them with a passion. The truth is that singletons aren't inherently bad if they're used correctly. The goal of the singleton pattern is to ensure only one instance of a class is alive at any one time. That, however, is not the goal many developers have in mind when using singletons.

Singletons are very much like the good things in life; they're not bad if used in moderation. The next time you're about to create a singleton, consider your motivation for creating it. Is it convenience? Then you should not create that singleton. Period. That's the simple rule I apply.

Dependency Injection

My favorite quote about dependency injection is a quote by James Shore. It summarizes much of the confusion that surrounds dependency injection.

> "Dependency Injection" is a 25-dollar term for a 5-cent concept. â€" James Shore

When I first heard about dependency injection, I figured it was a technique too advanced for my needs at that time. I could do without dependency injection, whatever it was.

What Is It

I later learned that, if boiled down to its bare essentials, dependency injection is a simple concept. James Shore offers a succinct and straightforward definition of dependency injection.

> Dependency injection means giving an object its instance variables. Really. That's it. â€" James Shore

For developers new to dependency injection, it's important to learn the basics before relying on a framework or library. Start simple. Chances are that you already use dependency injection without realizing it.

Dependency injection is nothing more than injecting dependencies into an object instead of tasking the object with the responsibility of creating its dependencies. Or, as James Shore puts it, you give an object its instance variables instead of creating them in the object.

An Example

In the example below, we define a `UIViewController` subclass that declares a property, `requestManager`, of type `RequestManager?`.

```
1 import UIKit
2
3 class ViewController: UIViewController {
4
5     var requestManager: RequestManager?
6
7 }
```

We can set the value of the `requestManager` property one of two ways.

Without Dependency Injection

The first option is to task the `ViewController` class with the instantiation of the `RequestManager` instance. We can make the property lazy or initialize the request manager in the view controller's initializer. That's not the

point, though. The point is that the view controller is in charge of creating the `RequestManager` instance.

```
import UIKit

class ViewController: UIViewController {

    var requestManager: RequestManager? = RequestManager()

}
```

This means that the `ViewController` class not only knows about the behavior of the `RequestManager` class. It also knows about its instantiation. That's a subtle but important detail.

With Dependency Injection

In the second option, we inject the `RequestManager` instance into the `ViewController` instance. Even though the result may appear identical, it isn't. By injecting the request manager, the view controller doesn't know how to instantiate the request manager.

```
// Initialize View Controller
let viewController = ViewController()

// Configure View Controller
viewController.requestManager = RequestManager()
```

Many developers immediately discard the second option because it is cumbersome and unnecessarily complex. But if you consider the benefits, dependency injection becomes more appealing.

Another Example

Let me show you another example to emphasize the point I made in the previous section. Take a look at the following example.

```
protocol Serializer {

    func serialize(data: AnyObject) -> NSData?

}
```

```
class RequestSerializer: Serializer {

    func serialize(data: AnyObject) -> NSData? {
        ...
    }

}
```

```
class DataManager {

    var serializer: Serializer? = RequestSerializer()

}
```

The `DataManager` class has a property, `serializer`, of type `Serializer?`. In this example, `Serializer` is a protocol. The `DataManager` class is in charge of instantiating an instance of a type that conforms to the `Serializer` protocol, the `RequestSerializer` class in this example.

Should the `DataManager` class know how to instantiate an object of type `Serializer`? Take a look at the following example to show the power of protocols and dependency injection.

```
// Initialize Data Manager
let dataManager = DataManager()

// Configure Data Manager
dataManager.serializer = RequestSerializer()
```

The `DataManager` class is no longer in charge of instantiating the `RequestSerializer` class. It no longer assigns a value to its `serializer` property. In fact, we can replace `RequestSerializer` with another type as long as it conforms to the `Serializer` protocol. The `DataMananger` no longer knows or cares about these details.

What You Gain

I hope the above examples have at least captured your attention. Let me list a few additional benefits of dependency injection.

Transparency

By injecting the dependencies of an object, the responsibilities and requirements of a class or structure become more clear and more transparent. By injecting a request manager into a view controller, we understand that the view controller depends on the request manager *and* we can assume the view controller is responsible for request managing and/or handling.

Testing

Unit testing is so much easier with dependency injection. Dependency injection allows developers to substitute an object's dependencies with

mock objects, making unit tests easier to set up and isolate behavior.

```
class MockSerializer: Serializer {

    func serialize(data: AnyObject) -> NSData? {
        ...
    }

}
```

```
// Initialize Data Manager
let dataManager = DataManager()

// Configure Data Manager
dataManager.serializer = MockSerializer()
```

Separation of Concerns

As I mentioned and illustrated earlier, another subtle benefit of dependency injection is a stricter separation of concerns. The `DataManager` class in the above example isn't responsible for instantiating the `RequestSerializer` instance. It doesn't need to know how to do this.

Even though the `DataManager` class is concerned with the behavior of its serializer, it isn't, and shouldn't be, concerned with its instantiation. What if the `RequestManager` of the first example also has a number of dependencies. Should the `ViewController` be aware of those dependencies too?

Coupling

The example with the `DataManager` class illustrated how the use of protocols and dependency injection can reduce coupling in a project. Protocols are incredibly useful and versatile in Swift. This is one scenario in which protocols really shine. Later in this chapter, I write more about protocols and protocol-oriented programming.

Types

Most developers consider three forms or types of dependency injection:

- dependency injection through an initializer
- dependency injection using properties
- dependency injection in methods

These types shouldn't be considered equal, though. Let me list the pros and cons of each type.

Initializer

I personally prefer to pass dependencies during the initialization of an object because this has several key benefits. The most important benefit is that dependencies passed in during initialization can be made immutable. This is very easy to do in Swift by declaring the properties for the dependencies as constants. Take a look at the below example.

```
class DataManager {

    private let serializer: Serializer!

    init(serializer: Serializer) {
        self.serializer = serializer
    }

}
```

```
// Initialize Request Serializer
let serializer = RequestSerializer()

// Initialize Data Manager
let dataManager = DataManager(serializer: serializer)
```

The only way to set the `serializer` property is by passing it as an argument during initialization. The `init(serializer:)` method is the designated initializer and guarantees that the `DataManager` instance is correctly configured. Another benefit is that the `serializer` property cannot be mutated.

Because we're required to pass the serializer as an argument during initialization, the designated initializer clearly shows what the dependencies of the `DataManager` class are.

Internal or Public Properties

Dependencies can also be injected by declaring an internal or public property on the class or structure that requires the dependency. This may seem convenient, but it adds a loophole in that the dependency can be modified or replaced. In other words, the dependency isn't immutable.

```
import UIKit

class ViewController: UIViewController {

```

```
    var requestManager: RequestManager?

}
```

```
// Initialize View Controller
let viewController = ViewController()

// Configure View Controller
viewController.requestManager = RequestManager()
```

Methods

Dependencies can also be injected whenever they're needed. This is easy to do by declaring a method that accepts the dependency as a parameter. In the example below, the serializer isn't a property on the `DataManager` class. Instead, the serializer is injected as an argument of the `serializeRequest(_:withSerializer:)` method.

```
class DataManager {

    func serializeRequest(request: Request, withSerializer serialize\
r: Serializer) -> NSData? {
        ...
    }

}
```

Even though the `DataManager` class loses some control over the dependency, the serializer, this type of dependency injection introduces flexibility. Depending on the use case, we can choose what type of serializer to pass into `serializeRequest(_:withSerializer:)`.

Singletons

Earlier in this chapter, I showed you how dependency injection can be used to eliminate the need for singletons in a project. I'm not a fan of the singleton pattern and I avoid it whenever possible. Even though I don't consider the singleton pattern an anti-pattern, I believe they should be used very, very sparingly. The singleton pattern increases coupling whereas dependency injection reduces coupling.

Too often, developers use the singleton pattern because it's an easy solution to a, often trivial, problem. Dependency injection, however, adds clarity to a project. By injecting dependencies during the initialization of an object, it becomes clear what dependencies the target class or structure has and it also reveals some of the object's responsibilities.

Dependency injection is one of my favorite patterns because it helps me stay on top of complex projects. This pattern has so many benefits. The only drawback I can think of is the need for a few more lines of code, but that's a drawback I'm happy to accept.

References, Delegation, and Notifications

A typical Cocoa application is composed of dozens and dozens of objects, working together to make your application tick. To get the job done, these objects need the ability to talk to each other. In this section, we take a look at three common patterns that enable objects to communicate with one another. We also discuss when to use which pattern and, more importantly, when to avoid a particular pattern.

References

On iOS, a view is typically owned and managed by a view controller. This is textbook Model-View-Controller. The view controller keeps a reference to the view it manages and talks to it directly.

Even though there's nothing wrong with this approach, it's important to understand that it introduces **tight coupling**. The implementation of the view controller is tightly coupled to that of the view. The opposite isn't true, though. The view is unaware of the view controller and, as a result, its implementation isn't coupled and doesn't depend on that of the view controller.

Keeping a reference to an object is the most direct way of communication. Because it introduces coupling between objects, it cannot, or should not, be used in every context.

Delegation

At first glance, delegation appears to be similar to the first approach. The key difference is that the objects are **loosely coupled**. The `UITableViewDelegate` protocol is a good example of delegation.

Like any other view, a table view is in charge of responding to user interaction. Even though a table view can detect user interaction, it

doesn’t know how to interpret the touch events. This seeming disadvantage is what makes table views reusable components.

A table view informs its delegate about the touch events it detects. It’s the responsibility of the table view’s delegate, a view controller, for example, to handle table view interactions. A table view is loosely coupled to its delegate through the `UITableViewDelegate` protocol. The delegate object is also loosely coupled to the table view. All it needs to do is conform to the `UITableViewDelegate` protocol by implementing the methods defined by the protocol.

Any class instance conforming to the `UITableViewDelegate` protocol can act as the table view’s delegate. The table view doesn’t care and doesn’t need to know which object acts as the table view’s delegate as long as it conforms to the `UITableViewDelegate` protocol.

This is an example of **protocol-oriented programming**, which we cover later in this chapter. As the name implies, the table view **delegates** tasks to the delegate object. Delegation is a very common design pattern in Cocoa applications and frameworks.

Notifications

Ideally, delegation is used to enable **unidirectional** communication **between two objects**. What if an object has a message that’s of interest to two, three, or dozens of objects? For Cocoa applications, the recommended approach is posting a notification through a notification center.

The `NotificationCenter` class, defined in the **Foundation** framework, provides an interface for broadcasting messages within a Cocoa application. To make this possible, two elements need to be in place:

- an object posting the notification
- one or more objects interested in the notification

An object needs to tell a notification center that it’s interested in receiving notifications with a particular name. It can optionally specify which objects they’d like to receive notifications from. What does that look like in code?

```
let defaultCenter = NotificationCenter.default
defaultCenter.addObserver(self, selector: #selector(synchronizationD\
idFinish(_:)), name: Notification.Name.SynchronizationDidFinish, obj\
ect: nil)
```

The object that broadcasts the notification uses the same notification center to post a notification with a particular name. It can optionally include additional information in the form of a dictionary.

```
let defaultCenter = NotificationCenter.default
defaultCenter.post(name: Notification.Name.SynchronizationDidFinish,\
 object: self, userInfo: [ "success" : true ])
```

If an object is no longer interested in receiving notifications with a particular name, it simply removes itself as an observer.

```
NotificationCenter.default.removeObserver(self)
```

Be Careful

Notifications are an essential aspect of Cocoa development. Apple's frameworks define dozens and dozens of notifications for various purposes. This may give you the impression that notifications are a proven, reliable, and robust solution to send messages from one object to another. While that's true, there's a catch.

Coupling

We like loose coupling. Right? Tight coupling is bad. This is true, but how loosely coupled is the notification pattern. If you have an object that subscribes to the `UIApplicationDidEnterBackground` notification, you implicitly tie its implementation to the UIKit framework. There's nothing wrong with this, but it's important to consider this. You're implicitly adding a dependency to the UIKit framework.

Debugging

In my early days developing Cocoa applications, I made ample use of two patterns, singletons and notifications. Looking back it seems only natural. I was trying to solve common problems and I found two solutions that were easy to implement.

Notifications aren't bad and they're very useful at times, but it's important to understand that you pay for them in transparency.

The first project I started and successfully finished made heavy use of notifications. Debugging this project was a nightmare at times. Xcode's debugger can make it easy to debug common problems, but notifications can make debugging much more complex and tedious.

A notification sent by one object can trigger a response in a distant object in your project. The only link between these objects is the notification.

If You Decide to Use Notifications

If you decide that notifications are the right solution to a problem, then, by all means, use them. However, I encourage you to adopt a handful of good practices if you do. Don't use string literals to name your notifications. Ever. Do yourself a favor and create an extension for `Notification` and follow Apple's lead. You can use your own naming scheme, but I advise you to use constants for notification names.

```
1 import Foundation
2
3 extension Notification.Name {
4
5     static let DidResetStatistics = Notification.Name(rawValue: "Did\
6 ResetStatistics")
7
8 }
```

Use notifications sparingly and only if there's no other alternative that's straightforward to implement. For example, don't use notifications instead of delegation. If you only intend to notify one object, then delegation is usually the right choice, not notifications.

Ask yourself whether notifications are the right solution to solve the problem. Be honest with yourself. Do you use notifications because it's convenient or because it's the right solution to the problem you're solving?

That's what I truly enjoy about software development. There are recipes and there are tools, but it's up to you, the developer, to use the right combination. There are usually multiple solutions, but no two solutions are the same. One solution is often preferred over another.

What About Key-Value Observing?

Key-Value Observing, or KVO for short, is a solution that enables objects to observe properties of other objects. Key-Value Observing and Key-Value Coding are important aspects of Cocoa programming. They're heavily used by bindings on macOS, for example. That said, it's important to understand that KVO is a passive form of communication between objects.

When To Use What

References

Object references make sense if both objects have a clear or logical relationship with one another. A view controller, for example, keeps a reference to the view it manages. It's important to understand that the owner takes responsibility for the lifetime of the object it owns. This isn't true for delegation.

Delegation

Developers new to Cocoa are often confused by delegation and notifications. There's no need to be confused, though. Delegation should only be used if one object needs to talk to another object. It's a one-to-one relationship and the communication is often unidirectional, from the delegating object to its delegate.

Delegation works best with protocols. A delegate object, the object being talked to, can be any object as long as it conforms to the delegate protocol. Remember how a table view talks to its delegate. It doesn't know which object it delegates tasks to. It only knows that it conforms to the `UITableViewDelegate` protocol. As a result, delegation promotes a loosely coupled object graph.

Notifications

Notifications are used if an object wants to notify other objects of an event. The sender of the notification doesn't care which objects subscribe to the notification or how they handle the broadcast. Even though this may seem the best solution since objects are seemingly uncoupled, it introduces a subtle form of coupling as I mentioned earlier in this chapter.

One Solution to Rule Them All

Great. I'm going to stick with notifications. *For everything*. It offers the most flexibility, well, not exactly. It's true that notifications are great for communication between objects that are unaware of each other.

As I wrote earlier, an application that relies heavily on notifications quickly becomes a nightmare to maintain and debug. Remember to only use notifications if you have no other option. Notifications aren't bad, but they shouldn't be overused either. They're no cure-all.

Closures as an Alternative to Delegation

Even though I like using delegation in Cocoa applications, it can sometimes become a bit overwhelming. Large applications can end up with dozens of delegate protocols, with some of them defining only a single method.

An alternative approach is to use closures (or blocks in Objective-C). I only recommend this alternative as a substitute for compact delegate protocols. Let's look at an example to better understand how this works.

Imagine an application that manages a list of items. The user can add items to the list by tapping a button in the navigation bar. Each item is an instance of the `Item` structure.

```
struct Item {

    var title: String
    var content: String

}
```

To add an item, the user taps a button in the navigation bar. This action instantiates an instance of the `AddItemViewController` class. The class declares a `didAddItem` property of type `DidAddItemHandler?`, a type alias for a closure that accepts an `Item` instance as an argument. This closure is invoked when the user creates and saves a new item.

```
class AddItemViewController: UIViewController {

    typealias DidAddItemHandler = (Item) -> ()

    var didAddItem: DidAddItemHandler?

    private var item: Item?

    @IBAction func save(_ sender: AnyObject) {
```

```
        if let item = item, let didAddItem = didAddItem {
            didAddItem(item)
        }
    }

}
```

The list of items is managed by an instance of the `ItemsViewController` class. This class implements the `addItem(_:)` action, which is linked to the button in the navigation bar. Tapping the button instantiates the `AddItemViewController` instance.

```
class ItemsViewController: UIViewController {

    var items = [Item]()

    @IBAction func addItem(_ sender: AnyObject) {
        // Initialize View Controller
        let viewController = AddItemViewController()

        // Configure View Controller
        viewController.didAddItem = { (item) in
            // Add Item to List
            self.items.append(item)
        }

        present(viewController, animated: true)
    }

}
```

Before presenting the add item view controller, we set its `didAddItem` property. The closure accepts an `Item` instance as its only argument and that enables the items view controller to add the new item to the array of items it manages. Not only removes this pattern the overhead of creating and implementing a delegate protocol, it also keeps related code together.

Singletons and Object References

There’s nothing wrong with object references, but it’s important to keep track which objects know about which objects. When object references are combined with singletons, you need to be extra careful to avoid getting yourself onto a slippery slope.

From the moment singletons keep references to other singletons, well, I hope by now that you know where that ends. Singletons aren’t bad, but you need to be careful when you use the singleton pattern. I avoid it by default and only use it when there’s no other option.

Reactive Programming

Later in this chapter, I talk about reactive programming. If you feel comfortable using the Cocoa SDK and the Swift language, then it may be time to look into reactive programming. It may seem daunting at first, but it's going to change how you think about code and software development.

Master These Patterns

Object references, delegation, and notifications are very common design patterns on iOS, tvOS, macOS, and watchOS. The Cocoa frameworks make heavy use of these patterns and it's important to understand how they work and when to use them. Even if you never define a delegate protocol yourself, you inevitably need to use them if you build Cocoa applications. And the same applies to notifications. And remember, this is no rocket science.

6 Protocol-Oriented Programming

Object-oriented programming, or OOP, is a widespread paradigm most developers are familiar with. The paradigm has been around for decades and it's proven its value in software development.

Because object-oriented programming is baked into the Cocoa SDK, it's impossible to create a Cocoa application without it. It should come to no surprise then that it's the default choice for most developers.

Not long after the introduction of the Swift language, a seemingly new paradigm made its way into the Swift community, **protocol-oriented programming**, or POP. What is it and should you care? Spoiler. You should.

But First This

Before I continue, I need to tell you about Dave Abrahams. Dave works at Apple and is a member of the Swift Core Team. A few years ago, at WWDC 2015, Dave gave an amazing presentation that made waves in the Swift community. In his presentation, Protocol-Oriented Programming in Swift, Dave challenges, with a dash of humor, the defaults most of us are used to.

I strongly encourage you to take some time to watch Dave's presentation about protocol-oriented programming. He does a much better job explaining what protocol-oriented programming is and how you can benefit from it. Dave also zooms in on some of the key driving forces of the Swift language. Like Dave emphasizes in his talk "Swift is a protocol-oriented programming language." and "At its heart, Swift is protocol-oriented."

What Is It

Let me start with a bit of context. The idea isn't as difficult to grasp as you might think. Most of us are used to the obvious benefits of object-oriented programming, such as encapsulation, access control, and inheritance. These are features we love and usually take for granted.

The truth is that object-oriented programming comes at a cost. We pay a price for the conveniences it offers. Object-oriented programming has its downsides and, without realizing it, we fight them on an almost daily basis.

What are some of those downsides? Inheritance is great, but you need to carefully choose the superclass your class inherits from. It's not always clear which methods a subclass can override and which ones it should override.

Inheritance also means that subclasses inherit stored properties from their superclass. That's a good thing. No? Complex class hierarchies can add a lot of cruft to your classes, and, because Swift requires you to provide an initial value for every stored property, initialization can be messy.

Classes are reference types and, while this is often exactly what you want, it can lead to complex problems that are hard to debug. Shared mutable state often leads to complexities that bite you at some point.

Is Protocol-Oriented Programming Any Better

Protocol-oriented programming isn't necessarily solving the shortcomings of object-oriented programming, but it's a better alternative in many

scenarios. Dave presents a detailed example in his presentation, highlighting the benefit of protocol-oriented programming in Swift.

Protocol-oriented programming starts with a different mindset. Instead of creating class hierarchies, you create types that conform to protocols. A protocol is nothing more than an interface, defining a blueprint of properties and methods. A type can conform to a protocol by implementing its interface.

Flexibility

Using protocols instead of inheritance has its pros and cons. You probably know the advantages of inheritance. What does protocol-oriented programming gain us?

In Swift, a class has only one superclass while a type can conform to multiple protocols. While value types don't support inheritance, structs and enums can conform to protocols.

Transparency and Abstraction

Because a protocol defines an interface, it's immediately obvious to a type conforming to a protocol which properties and methods it should implement. This isn't always true if you're subclassing a class.

As Dave points out in his presentation, protocol-oriented programming is often a better solution for abstraction. You can find an example of this in Mastering MVVM With Swift. We use protocol-oriented programming to add a layer of abstraction between the view model and the table view cell it configures. The view model conforms to a protocol and that allows the table view cell to configure itself with the view model without the need for the table view cell to know about the view model's existence, definition, or interface.

Be Careful

While protocols may seem convenient and flexible, they can also complicate your codebase. I've worked on projects that were difficult to understand because of the overuse of protocols.

Chris Eidhof wrote about this problem a few months ago. He illustrates the versatility of a protocol-oriented design, but he also emphasizes that it shouldn't be your default choice.

> [P]rotocols can be very useful. But don't start with a protocol just for the sake of protocol-oriented programming. Start by looking at your problem, and try to solve it in the simplest way possible. Let the problem drive the solution, not the other way around. Protocol-oriented programming isn't inherently good or bad. Just like any other technique (functional programming, OO, dependency injection, subclassing) it can be used to solve a problem, and we should try to pick the right tool for the job. Sometimes that's a protocol, but often, there's a simpler way.

Remember what I wrote in the chapter on refactoring. You solve a complex problem by implementing the simplest possible solution you can come up with. With the problem solved, it's time to optimize your solution.

Watch Dave's Presentation

Don't forget to watch Dave's presentation. I've watched it several times and you should too. It's a fun presentation that teaches you a lot about the Swift language, including protocol-oriented programming.

7 Reactive Programming

As a developer, you'll have several *aha* moments in your career. The discovery of reactive programming was one of those moments for me. When I first heard about reactive programming, I was skeptical. It looked exotic and magical, and I discarded it almost immediately. A few years later, RxSwift crossed my path and it clicked.

It's true that there's a learning curve, especially if you read random tutorials on the web. But once it clicks, you see the benefits and what reactive programming brings to the table.

What Is It

Trying to find a definition of reactive programming that makes sense is the first challenge you'll face. Let's keep it very simple and stick with André Staltz's definition.

> Reactive programming is programming with asynchronous data streams. — André Staltz

André has a comprehensive guide on reactive programming that you should read if you're interested in adopting reactive programming. As André writes, reactive programming is nothing more than interacting with streams of data. These are commonly referred to as **observable sequences**, or observables for short.

Reactive programming isn't built into Swift. As a result, several frameworks and libraries have emerged for Swift and Cocoa over the years. The most popular solutions are ReactiveCocoa and RxSwift. I've chosen for RxSwift because it's the official extension for Swift of the ReactiveX API. It's up to you to decide which implementation you prefer.

Why, Why, Why

As I mentioned earlier, I was skeptical when I first heard and read about reactive programming. I didn't see *why* I should invest time in yet another technology. Objective-C and Swift had served me well for many years. It didn't click for me at the time. Yet reactive programming has become an integral part of my workflow. Why is that? What changed my mind?

Simplification

Applications that respond to events, such as user interaction, tend to become complex rather quickly. Let's face it, asynchronous programming isn't easy. A request is sent to a remote API, at some point you receive a response, and that response is used in a distant unrelated section of your application. It can be a challenge to implement even the simplest tasks.

Reactive programming aims to make this easier. You no longer drown in callbacks. You simply work with streams of data that you monitor, compose, and respond to. You shift from imperative programming to reactive programming.

Removing State

While it's this simplification that draws most developers to reactive programming, you receive another important benefit if you decide to adopt reactive programming. Because you're observing and responding to streams of values, you no longer maintain state. Why is that important? Why is this a big deal?

Almost every application manages state in some form or shape. If you're developing a mobile application, then you need to reflect some of that state in the user interface. While this may seem easy on paper, it's much harder than it looks. I'm sure you know what I mean if you're working on projects that have a bit of complexity to them.

Reactive programming makes this much easier. For the past few months, I've been working on an application with a complex timer engine. This would have been a nightmare without reactive programming. Instead of keeping track of the state of the timer, the time, the current segment, and on and on, the application observes and responds to the values of several observable sequences. It's amazing.

Coupling

Reactive programming promotes loose coupling of objects. Any object can observe an observable sequence and it only needs to keep a reference to that stream of values. That's the only aspect it's interested in.

The delegation pattern is very easy to replace with reactive programming. Instead of keeping a reference to a delegate object, the delegating object manages an observable sequence, emitting values when appropriate. Any other object that's interested in the values emitted by that observable sequence can subscribe to it.

All or Nothing

Developers unfamiliar with reactive programming wrongly believe that reactive programming requires an all-or-nothing approach. That isn't true, though. Once you've included RxSwift and RxCocoa in one of your projects, it's up to you to decide where, how, and why you want to use it.

The application we build in Mastering MVVM With Swift is a good example. We only use reactive programming in a handful of classes. The rest of the project is unaware of RxSwift and RxCocoa.

Is This For You

I encourage every developer to have a look at reactive programming. However, if you're new to software development, then my advice is to focus on your foundation first. Remember what I wrote earlier in this book. Reactive programming can wait.

Model-View-ViewModel

In Mastering MVVM With Swift, I illustrate the powerful combination of the Model-View-ViewModel pattern and reactive programming. The Model-View-ViewModel pattern works best with a bindings solution, and reactive programming offers just that.

Remember that it doesn't need to be RxSwift and RxCocoa. You can use the solution that best fits your style and preferences.

Part 2: Swift

1 Learn Swift With an Open Mind

Developers making the switch from Objective-C to Swift tend to translate Objective-C into Swift. They take what they know about Cocoa development and apply it to Swift. Unfortunately, this often results in frustration and sometimes even an aversion towards the Swift language.

It's easy enough to understand why developers take this approach. People learning a new spoken language tend to look for the patterns and constructs they're already familiar with. While this is a common phase in the learning process, it emphasizes the importance of a carefully chosen learning trajectory. The same applies to picking up a new programming language.

> Developers new to Swift often take these commonalities to think that Swift and Objective-C are very similar while they're not.

Because Swift is so different from Objective-C, you need to take a step backward. You need to unlearn what you know about Objective-C and start with a clean slate and an open mind. Swift and Objective-C differ more than they're alike. It's true that a Swift application runs in the Objective-C runtime and that the languages are interoperable, they understand one another. But developers new to Swift often take these commonalities to think that Swift and Objective-C are very similar while they're not.

Reference Types and Value Types

While there's nothing inherently wrong with classes and inheritance, Swift implicitly encourages the use of value types (tuples, structures, and enumerations) instead of reference types (classes). Several months ago, I came across a talk from Andy Matuschak about this topic. It's a great introduction to the advantages value types have over reference types.

The Swift Standard Library also embraces value types. Strings, arrays, dictionaries, and sets, for example, are value types in Swift. This is very different from their Foundation counterparts, `NSString`, `NSArray`, `NSDictionary`, and `NSSet`.

Protocol-Oriented Programming

Even though structures and enumerations in Swift are more powerful than their Objective-C counterparts, they don't support inheritance. This is often seen as a missing feature and scares many developers away from value types. Developers new to Swift and accustomed to object-oriented programming look for inheritance and find it in classes, that is, reference types.

Earlier in the book, I emphasize that the lack of inheritance isn't a problem. Protocols and value types are a potent combination, maybe even more so than reference types and inheritance.

Apple emphasizes that protocol-oriented programming is a pattern developers should embrace in Swift. I encourage you to watch Protocol-Oriented Programming in Swift to understand what it is and how you can apply it in Swift. Dave Abrahams does a tremendous job explaining why protocol-oriented programming is a natural fit for Swift.

Classes and inheritance remain important in Swift. They're not the enemy. The Cocoa frameworks are for the most part powered by Objective-C and that means classes and inheritance are here to stay, at least for the foreseeable future.

As a Cocoa developer, you continue to use classes and create subclasses. But it's equally important to understand the ideas that power Swift and how you can use them in your projects. In his presentation, Dave repeatedly draws attention to Swift being a protocol-oriented language.

Type Safety

Type safety is a cornerstone of the Swift programming language. As a developer, Swift expects you to be clear about the types of the variables and constants you use. The advantage is that common mistakes, such as

passing a value of the wrong type to a method, can be caught by the compiler.

Developers coming from Objective-C often struggle with Swift's strict type safety rules. This ties in with another concept of Swift, **optionals**. Optionals are seen as an obstacle, an unavoidable side effect of Swift's type safety. What happened to sending messages to `nil` in Objective-C? What's wrong with that?

Optionals may indeed feel as obstacles when you first start experimenting with Swift. As time passes and you become more familiar with the language in its entirety, the pieces of the puzzle start to come together and you come to appreciate optionals for what they represent and the problem they solve. In combination with optional binding and optional chaining, optionals start to fit into that proverbial puzzle.

I agree that optionals can feel clunky if you're interacting with an Objective-C API that wasn't built with Swift or optionals in mind. But I wouldn't want to use Swift without them. Optionals are such a nice feature of Swift and I've come to appreciate them. In fact, every time I take a trip down memory lane when I need to fix a bug in Objective-C, I miss them. Optionals clearly express what's happening.

If you're still not sure about optionals, then I suggest you read the chapter about optionals in this book. Give them the benefit of the doubt for now. Use them for a few weeks, avoid the exclamation mark, and see how you feel about them in a month.

Best Practices

The more you read about Swift and the longer you spend time in the Swift community, the more you realize that Swift is still a very young language, especially if you compare the language with C and Objective-C. A common problem developers face is the lack of best practices. They look for strategies to solve common problems and struggle to find them. Best practices are slowly taking form as more people use the language in real-world scenarios.

This isn't surprising. The Swift community is still exploring the language, finding out what's possible and how things can or should be done. The

language is still developing at a rapid pace and the open sourcing of the language is another component that drives this *swift* evolution.

Several best practices are slowly taking shape, such as the adoption of protocol-oriented programming, the value of immutability, and the use of value types over reference types.

Forget What You Know

My advice is to approach Swift with an open mind. Try to forget what you know. What you know has value, but it may slow you down or it may lead to frustration. Explore the language, become familiar with the foundations it’s built on, and learn from others.

2 Be Critical

The primary goal of Cocoacasts is to help developers become better at their craft. It can be challenging to put that into words. But there's one trait I feel every good developer should have, being critical. This means being critical of your own work, but also of that of others.

If you spend time reading blogs, watching videos, or browsing social media, you inevitably pick up new techniques, tips, and tricks that can help you in your work. It's often tempting to absorb this information as gospel if the source is a respected developer or an authority in their field.

Even though this is a natural response, it sometimes hints at a lack of confidence. My advice is to always be critical. The responses I get on my tutorials and videos are most often questions. But there's a number of developers that send me their thoughts on the subject or they challenge me with an alternative. That's a healthy attitude. The reason can be arrogance, but, more often, it's just someone being critical.

Stack Overflow is a blessing for the developer community. I don't think there's a developer that doesn't occasionally visit Stack Overflow for answers. But it's important to be critical of what you read. Stack Overflow has a reliable voting system for making sure answers are correct and tested, but that doesn't mean you should take what you read on Stack Overflow as the truth.

Whenever you read a solution to a problem you're having it's important to take a closer look at that solution before embracing it. Make sure you understand the solution. Does the solution make sense? How could you improve the solution?

The tips and practices I describe in this part have become an integral part of my workflow. Some of them are opinionated while others are considered good practices for Swift development. I leave it up to you to decide whether they're worth adopting. Remember to always be critical.

3 Embracing Optionals

Type safety is a fundamental concept of the Swift programming language and optionals neatly tie into Swift's strict type safety rules. The concept underlying optionals is simple, an optional has a value or it doesn't.

Developers new to this concept often struggle with optionals. They see optionals as an obstacle and a direct consequence of the strict type safety rules imposed by the language. This struggle usually dissipates after spending some time with the language and after discovering Swift's constructs for safely interacting with optionals, such as optional binding and optional chaining.

Warning Sign

Before the struggle ends, though, brute force is used to attack the problem that optionals appear to be. After discovering the exclamation mark, every optional is considered a nail that needs to be hit with the proverbial hammer. Forced unwrapping and implicitly unwrapped optionals are temporarily seen as the solution to the problem.

The exclamation mark is in Swift what you may intuitively expect it to be, a warning sign. If you use the exclamation mark, you're about to perform an operation that may backfire. Forced unwrapping an optional that doesn't contain a value throws a runtime error, crashing your application. The exclamation mark warned you this could happen, though.

I recently received an email from a reader about the use of the exclamation mark. The company she works for has a simple policy, "Don't use the exclamation mark." This may at times result in code that's a bit verbose, but the benefits are obvious. The code they write is safe and it's consistent.

Elegance and Beauty

The beauty of optionals is easily overlooked if not paid attention to. Optionals embody a concept that's easy to grasp and that's what makes it powerful. If you're given a non-optional, you're given the promise that it has a value, it's anything but `nil`. In computing, that's a bold promise.

> If you know that a constant or variable is guaranteed to have a value, you're also given freedom.

If you know that a constant or variable is guaranteed to have a value, you're also given freedom. A pointer in Objective-C, for example, isn't guaranteed to not be `nil`. While it may be pointing to a valid object now, that may change later. No problem. You can safely send messages to `nil` in Objective-C. That's wonderful, but how often has this been an advantage? Think about it.

If you're sending a message to `nil`, it probably means that you were expecting something to happen that didn't happen. Is that a better solution than optionals?

> If you're sending a message to `nil`, it probably means that you were expecting something to happen that didn't happen.

Swift to the Rescue

Swift doesn't leave developers out in the cold. The language offers a number of constructs for working with optionals, such as optional binding and optional chaining.

Optional binding is used to safely unwrap the value that's stored in an optional. If the optional has a value, then that value is bound to a constant that's available in the `if` clause.

```
1 if let message = message {
2     print(message)
3 } else {
4     print("no value")
5 }
```

This solution is more elegant than verifying whether `message` has a value and then forced unwrap the optional using the exclamation mark.

```
if message != nil {
    print(message!)
} else {
    print("no value")
}
```

Optional binding is flexible and powerful. Take a look at the following example.

```
if let payload = payload,
   let message = payload["message"] {
    print(message)
}
```

Optional chaining is an elegant alternative to forced unwrapping optionals. Instead of forced unwrapping an optional, you use optional chaining to access a property or invoke a method on the value stored in the optional.

```
label?.textColor = UIColor.blackColor()
```

The `textColor` property is only set if `label`, an optional, has a value. The question mark is used to safely set the `textColor` property through optional chaining.

At times, I use Swift's nil-coalescing operator when I'm faced with an optional. This can be useful to elegantly access the value of an optional or provide a default value.

```
func date(for profile: Profile) -> Date {
    return profile.createdAt ?? Date()
}
```

The `createdAt` property of the `Profile` instance is of type `Date?`. The `date(for:)` function always returns a `Date` instance by defaulting to the current date and time.

Embrace Optionals

Even though it may seem as if optionals are a solution to work around Swift's strict type safety rules, they're complementing those rules. Optionals get rid of unsafe pointers and offer guarantees that make your code safer and easier to debug.

> Optionals get rid of unsafe pointers and offer guarantees that make your code safer and easier to debug.

While it may seem great that you can send messages to `nil` in Objective-C, it's an empty promise. If you consider your application not crashing when you send a message to a `nil` pointer, then think again. Also, consider that optionals can be used for any type while `nil` only applies to pointers in Objective-C.

A few days ago, I had to open up an old project that was primarily written in Objective-C. The absence of type safety and optionals was immediately obvious. It may sound odd, but I miss optionals every time I need to work with Objective-C.

If you're serious about Swift, then it's important that you become familiar with optionals, which problems they solve, and embrace them in the code you write.

4 Mind the Exclamation Mark

I’m sure the exclamation mark is a necessary aspect of the Swift language, but I wouldn’t blink an eye if it were removed from the language tomorrow. The exclamation mark is a warning sign, but too many developers ignore this and use it because it’s more convenient. Note that I’m not referring to the logical `NOT` operator. That’s a perfectly valid use of the exclamation mark.

As I mention later in the book, the use of the exclamation mark is often a code smell. Because you’re trading safety for convenience, it should be used sparingly.

Use Cases

The exclamation mark is used in several situations. The most common use is forced unwrapping the value stored in an optional. As you know, a runtime error is thrown if that optional doesn’t hold a value.

You also use the exclamation mark to declare an implicitly unwrapped optional or when you want to disable error propagation. The same rule applies. If you access the value of an implicitly unwrapped optional that doesn’t contain a value or a throwing function throws an error when error propagation is disabled, a runtime error is thrown.

Convenience and Laziness

Developers new to Swift often resort to the exclamation mark because they’re unfamiliar with or confused by optionals. Most developers go through this phase, but I always recommend to get used to optionals sooner rather than later.

Those of us with a background in Objective-C usually have a harder time becoming familiar with optionals. We’re used to the fact that sending messages to `nil` doesn’t cause havoc in Objective-C. It does in Swift and that’s an unwelcome change.

Don’t Be Lazy

You probably heard the phrase “A lazy developer is a good developer.” This saying is true, but it depends on what’s understood by *lazy*. Many, many developers use the exclamation mark because it’s convenient. In other words, they’re being lazy. I’d like to say that laziness or convenience is never a valid reason for using the exclamation mark. Unfortunately, it’s difficult to argue with that explanation.

When I Use the Exclamation Mark

I use the exclamation mark in three situations, outlets, required properties, and resources.

Outlets

Outlets that should always have a value are declared as implicitly unwrapped optionals. As with every use of the exclamation mark, I only declare an outlet as an implicitly unwrapped optional to conveniently access its value. That’s the only reason. You can declare outlets as optionals. That’s perfectly fine too.

If an outlet is declared as an implicitly unwrapped optional and it doesn’t have a value when it’s accessed, then I made a mistake I need to fix.

Bear in mind that it’s possible to have optional outlets. In some projects, I create a `UIViewController` class that serves as the base class for other view controllers of the project. This class declares an optional table view outlet. Because not every view controller has a table view, it’s wise and necessary to declare the outlet as an optional.

Required Properties

Swift works fine with the Cocoa SDK, but there are some rough edges. You probably know that every stored property of a class or structure needs to have a valid initial value by the time an instance of the type is created. This is fine, but it presents some problems if you’re working with storyboards. If you’re using storyboards, then you’re not in charge of instantiating the view controllers of your application. This means that you can’t implement a custom initializer that accepts values for every stored property.

An approach that's often used is that you configure the destination view controller in the `prepare(for:sender:)` method of the source view controller. At that time, the destination view controller is already initialized and every stored property has an initial value. If you want to pass a model object to the view controller, you need to declare the property for that model object as an optional or, you guessed it, as an implicitly unwrapped optional.

As I mentioned earlier, the motivation for declaring a property as an implicitly unwrapped optional is always convenience. The reasoning behind this choice, however, is more nuanced. Let's assume you're implementing a view controller that displays the details of a note. You pass the note to the detail view controller. The detail view controller is useless if it doesn't have a valid note to work with. In other words, the `note` property of the detail view controller should always have a value. No exceptions.

Later in this book, I discuss fatal errors. In this situation, I prefer to throw a fatal error if the `note` property doesn't have a value because that should never happen in production. Unfortunately, I haven't found an elegant solution to implement this, which is why I use implicitly unwrapped optionals instead.

Resources

There's one other use case in which I use the exclamation mark. If the application needs to access a resource from its bundle, I don't want to deal with optionals. Why is that?

The resource, an image or a storyboard, should be present in the application bundle. If it isn't, then that means I have bigger problems to worry about. Take a look at this code snippet to better understand how I use this technique.

```
1 extension UIImage {
2
3     enum Icons {
4
5         enum Profile {
6
7             static let Segments = UIImage(named: "icon-profile-segme\
8 nts")
9             static let Configuration = UIImage(named: "icon-profile-\
```

```
configuration")

        }

    }

}
```

I also apply this technique to URLs that are hard-coded in the project.

```
enum Audio {

    static let Silence = Bundle.main.url(forResource: "rocknroll", w\
ithExtension: "mp3")!

}

enum API {

    static let BaseUrl = URL(string: "https://api.myawesomeapplicati\
on.co")!

}
```

A Personal Choice

While it’s a personal choice when you use the exclamation mark and what your motivation is, it’s important that understand why you’re using the exclamation mark and what the consequences are. The next time you append an exclamation mark to a variable, consider your motivation and the risk. Choose wisely and don’t be lazy.

5 Exclamation Marks and Fatal Errors

Fatal errors have a negative connotation and with reason. You should use them sparingly if you want to avoid having your application crash and burn at the slightest hiccup. Despite their negative undertone, fatal errors are an integral part of my workflow as I write elsewhere in this book.

Whenever I write or speak about my use of fatal errors, I usually see two types of responses. Developers unfamiliar with fatal errors and how they can be used safely are surprised and excited. They spot the benefits fatal errors can bring to a project. Can you guess what the second type of response sounds like?

> Why don’t you use an exclamation mark instead?

The suggestion to use an exclamation mark instead of throwing a fatal error is understandable. From a user’s perspective, the result is identical. But I’m not using fatal errors with the user in mind. I don’t throw a fatal error to crash the application when the user is using it. In an ideal scenario, a fatal error should only be thrown in development or when the application is being tested.

I agree that the user won’t appreciate my use of fatal errors if the application crashes the moment they’re about to best their previous high score. The thing is that I’m a developer and I look at code most of my working hours. And that’s exactly the reason I choose for fatal errors more frequently than I choose for the exclamation mark. Let me explain what I mean by that.

Clarity Over Subtleness

My biggest complaint with the exclamation mark is its subtleness. Ironically, plenty of developers use the exclamation mark for exactly that reason. It’s so easy to append an exclamation mark to a variable or a constant. It’s almost too easy. I understand why the Swift team has

chosen to support forced unwrapping and forced conversion using the `as!` operator, but I wouldn't shed a tear if both were removed from the Swift language tomorrow.

I agree that it can be frustrating to interact with an ancient Objective-C API that doesn't care about `nil` and optionals. Interacting with the file system, for example, can often lead you down a rabbit hole of optionals, indentation, and conditionals. But that's what it takes if you decide to write software in Swift.

Don't be lazy by appending an exclamation mark to a variable or a constant you're pretty sure will always contain a value. It will contain a value … most of the time … almost always. As the documentation explains, you should only force unwrap an optional if you're certain that it contains a value. I turn it around when I use fatal errors. A fatal error should be thrown if the application enters a state it didn't anticipate.

Choosing for Clarity

A common trait among developers is an obsession with simplicity and minimalism. Clean code is but one manifestation of this trait. By using fatal errors I choose for clarity. If the application throws a fatal error, I want to know about it. It's true that the exclamation mark will also do that for me. But I also want to know about it when I'm simply reading through my code.

An exclamation mark doesn't jump out, but a `guard` statement with a `fatalError()` call does. It immediately shows you that you know that a certain scenario should never happen and you guard against that.

Take a look at the following implementation of the `prepare(for:sender:)` method. This is a common pattern I use. If the user triggers the `Segue.SelectProfile` segue, the application expects the destination view controller to be of type `SelectProfileViewController`. It simply doesn't know how to respond if that isn't true hence the fatal error in the `else` clause of the `guard` statement. While it may look a bit verbose, it's clear and explicit.

```
1 // MARK: - Navigation
2
3 override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
```

```
    guard let identifier = segue.identifier else { return }

    switch identifier {
    case Segue.SelectProfile:
        guard let destination = segue.destination as? SelectProfileV\
iewController else {
            fatalError("Unexpected Destination View Controller for S\
egue")
        }

        ...
    default: break
    }
}
```

The alternative is to use the `as!` operator to forcefully convert the destination view controller to the type the application expects.

```
// MARK: - Navigation

override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    guard let identifier = segue.identifier else { return }

    switch identifier {
    case Segue.SelectProfile:
        let destination = segue.destination as! SelectProfileViewCon\
troller

        ...
    default: break
    }
}
```

I understand that this practice is a bit controversial, but I've seen its effectiveness. It's why I'm a big fan of this pattern. As I mentioned earlier in this chapter, the issue I have with the exclamation mark is that it isn't explicit enough, it's too subtle. It's easy to overlook it while browsing a codebase.

The difference between the use of fatal errors and the use of the exclamation mark is subtle. You could also say that the difference is easy to miss, which is exactly why I bring it up. Give it a try and let me know what you think.

6 Smelly Code

Pixelsync was the first iOS application I published on the App Store. For a first project, it was quite complex and far more challenging than I had anticipated. The project grew quickly as I added more features and maintainability quickly became an issue I couldn't ignore.

Looking back, the project was littered with anti-patterns, bad practices, and code smells. Adding features to a large, complex codebase is challenging if it lacks direction and structure.

Many developers start out this way and learn as they go. It's fine to make mistakes as long as you learn from your mistakes and find solutions that work better. In this chapter, I want to focus on common signs of code smell in Swift.

Forced Unwrapping and Conversion

Once you get used to optionals, you come to appreciate their value. Not only are optionals making your code safer, they also make your code more readable. An optional carries a message that says "I may not have a value. Be careful."

Developers new to Swift often see optionals as a hurdle, forcing them to jump through a bunch of unnecessary hoops. Optionals need to be unwrapped and that requires more code. There's a shortcut, though. It's possible to forced unwrap optionals. Take a look at the following example.

```
1 private func profileData(for launchOptions: [UIApplicationLaunchOpti\
2 onsKey: Any]?) -> [ProfileData]? {
3     let fileUrl = launchOptions![UIApplicationLaunchOptionsKey.url] \
4 as! URL
5     return profileData(at: fileUrl)
6 }
```

This looks fine. Right? We're certain that the dictionary of launch optionals isn't equal to `nil`, that the `UIApplicationLaunchOptionsKey.url`

is present, and that its value is of type `URL`.

It doesn't matter how certain you are, an optional should be treated with caution. Forced unwrapping or forced conversion is asking for trouble. As I discussed in an earlier chapter, there are situations in which you can use the `!` operator, but do not forced unwrap an optional to avoid a `guard` or an `if` statement or a few extra lines of code.

The updated example shows you how the `guard` statement can elegantly and safely handle optionals. The `guard` statement tells us what we expect, but, at the same time, it hints that we may not get what we expected.

```
1  private func profileData(for launchOptions: [UIApplicationLaunchOpti\
2  onsKey: Any]?) -> [ProfileData]? {
3      guard let launchOptions = launchOptions else {
4          return nil
5      }
6
7      guard let fileUrl = launchOptions[UIApplicationLaunchOptionsKey.\
8  url] as? URL else {
9          return nil
10     }
11
12     return profileData(at: fileUrl)
13 }
```

> If you find yourself using a `!` instead of an `?`, then stop for a moment and ask yourself whether there's a safer alternative.

Monster Classes

Andy Matuschak is a great speaker and a very good teacher. He once gave a wonderful talk at the Realm offices about refactoring view controllers. In the talk, Andy illustrates that view controllers are often taking on far too many responsibilities.

Some of us apply the Model-View-Controller pattern a bit too strictly. Did you know that it's fine to create classes that are not models, views, or controllers?

Some developers even argue that you should create fat models and skinny controllers. While that may be a bit too much responsibility for the model layer, I agree that controllers should be lean and lightweight. Of

course, it’s easy to stuff business logic in controllers and it takes a bit of extra work to refactor code that can belong in a separate class. But it’s often the right choice.

By isolating functionality, you put the view controllers of your project on a diet and make code easier to reuse. Why not extract a piece of functionality in an Objective-C category or a Swift extension.

If the implementation of a class exceeds 1000 lines of code, I know it’s time for a serious round of refactoring. Can you load 1000 lines of code in your working memory? That’s what it takes to work with a class. If you make changes to a class, you need to know how other parts are affected and that’s only possible if you know what the various parts of the class do.

Massive Methods

Monster classes very often harbor massive methods. Few things in a developer’s life are as frustrating as unit testing a method that spans dozens and dozens of lines. As I mention elsewhere in the book, functions and methods should be focused and to the point. A function or method should ideally have a singular focus.

Helper Methods

Most of my view controllers have a handful to a dozen helper methods. These helper methods are focused and help other methods be the same. Do the `viewDidLoad()` methods of your view controllers span dozens and dozens of lines? That’s a code smell.

I diligently try to keep methods short and succinct. Every view controller I implement that has any complexity to it has a `setupView()` and a `updateView()` method. I use these methods to factor any view configuration out of the `viewDidLoad()` method into these helper methods.

```
1 // MARK: - View Life Cycle
2
3 override func viewDidLoad() {
4     super.viewDidLoad()
5
6     // Set Title
7     title = NSLocalizedString("root_view_title", comment: "Root View\
8  Title")
9
```

```
10      // Setup View
11      setupView()
12
13      // Setup Notification Handling
14      setupNotificationHandling()
15  }
```

Property Observers

Configuring views isn't my favorite pastime. To remedy this, I often add a `didSet` property observer to an outlet to configure it when it's set. There are several benefits, one of them being that it keeps the configuration close to the outlet's declaration.

```
1  @IBOutlet var messageLabel: UILabel! {
2      didSet {
3          messageLabel.numberOfLines = 0
4          messageLabel.font = UIFont.Namaste.lightRegular
5          messageLabel.textColor = UIColor.Namaste.lightGray
6          messageLabel.text = NSLocalizedString("root_message_no_profi\
7  les", comment: "")
8      }
9  }
```

I'm sure several developers frown when they see this pattern in the wild. While I'm not sure, I believe I learned this trick from Natasha Murashev. Credit where credit is due.

Ambiguous Method Names

Massive methods aren't only hard to unit test, they almost always have names that make little or no sense. Because massive methods perform a slew of tasks, it's difficult to name them.

Massive methods usually have long names that don't reflect the implementation or they have a vague name that could mean anything. Break the method up into smaller, focused methods and give each a descriptive name. This small change immediately improves the testability of massive methods.

Ignoring Errors

It's great to see that error handling is tightly integrated into the Swift language. In Objective-C, it's easy to ignore errors, a bit too easy if you ask me. Have you ever seen something like this?

```
1  [managedObjectContext save:nil];
```

I don’t know any developer who enjoys error handling, but if you want to write code that works and applications that can recover when things go haywire, then you need to accept that error handling is part of the job. Ignoring errors also makes debugging more difficult. Putting your head in the sand isn’t the solution when things go wrong. Don’t worry, though. Later in this book, we explore error handling in a bit more detail.

Singletons

Singletons are great. They’re so useful that less experienced developers tend to overuse them. Been there, done that. I’m not joking when I say that singletons are great. Even though I don’t consider the singleton pattern an anti-pattern, sometimes it seems as if developers have forgotten how to pass an object by reference.

One of the negative side effects of singletons is tight coupling. Over the years, I have come across projects that were littered with singletons, singletons referencing other singletons. This can make refactoring a nightmare.

After removing the singletons from a project, it’s no longer intimidating, and it loses some of its complexity. By passing objects by reference, even if only one instance is alive at any one time, it becomes much clearer where the object is used and what role it plays.

Another benefit of keeping singletons to a minimum is testing. Singletons make testing more difficult.

There’s nothing wrong with passing objects by reference. Whenever you’re about to create a singleton, ask yourself whether passing the object by reference is an option. That should be your first choice. Always.

String Literals

String literals are smelly by default. They’re great in playgrounds, but how often do you, or should you, use string literals in a project? The only valid use cases I can think of are localization, constants, and logging.

I agree that creating constants or enums is tedious, but it pays off in the long run. You avoid typos, autocompletion kicks in, and it significantly

improves maintainability.

This certainly isn't a hard rule that I apply to the projects I work on, but I try to stick to it as much as possible. If the value of an object literal is used in more than one place, it's defined as a constant or enumeration, or, even better, put in a configuration file.

My Xcode color scheme highlights string literals in bright red. When I'm browsing through lines and lines of code, the string literals stand out immediately. If I see too much red, it's time for a clean-up.

```
14 class RootViewController: UIViewController {
15
16     fileprivate enum Segue {
17
18         static let Timer = "Timer"
19         static let Profiles = "Profiles"
20         static let AddProfile = "AddProfile"
21         static let Onboarding = "Onboarding"
22         static let Statistics = "Statistics"
23         static let ProfileSettings = "ProfileSettings"
24         static let ApplicationSettings = "ApplicationSettings"
25
26     }
27
28     // MARK: - Properties
29
30     @IBOutlet var messageLabel: UILabel! {
31         didSet {
32             messageLabel.numberOfLines = 0
33             messageLabel.font = UIFont.Namaste.lightRegular
34             messageLabel.textColor = UIColor.Namaste.lightGray
35             messageLabel.text = NSLocalizedString("root_message_no_profiles", comment: "")
36         }
37     }
38
```

String literals are highlighted in bright red.

Is Your Code Smelly

As I mentioned earlier, the goal of this chapter is to review the code you write. How can I improve this? What side effects does this implementation have? Should I forced unwrap this optional? It rarely pays off to take a shortcut.

7 Value Types and Reference Types

When talking about object-oriented programming, most of us intuitively think about classes. In Swift, however, things are a bit different. While you can continue to use classes, Swift has a few other tricks up its sleeve that can change the way you think about software development. This is probably the most important shift in mindset when working with Swift, especially if you're coming from a more traditional object-oriented programming language like Ruby or Objective-C.

What's the Fuss

The concept we tackle in this chapter is the differences between **value types** and **reference types** or, put differently, passing by value and passing by reference.

In Swift, instances of classes are passed by reference. This is similar to how classes are implemented in Ruby and Objective-C. It implies that an instance of a class can have several owners that share a copy.

Instances of structures and enumerations are passed by value. Every instance of a struct or enum has its own unique copy of data. And the same applies to tuples.

In the remainder of this chapter, I refer to instances of classes as objects and instances of structs and enums as values. This avoids unnecessary complexity.

An Example

It's important that you understand the above concept so let me explain this with an example.

```
1 class Employee {
2
3     var name = ""
4
5 }
6
```

```
var employee1 = Employee()
employee1.name = "Tom"

var employee2 = employee1
employee2.name = "Fred"

print(employee1.name) // Fred
print(employee2.name) // Fred
```

We declare a class, `Employee`, and create an instance, `employee1`. We set the `name` property of `employee1` to `Tom`. We then declare another variable, `employee2`, and assign `employee1` to it. We set the `name` property of `employee2` to `Fred` and print the names of both `Employee` instances. Because instances of classes are passed by reference, the value of the `name` property of both instances is equal to `Fred`. Does that make sense or does the result surprise you?

```
struct Employee {

    var name = ""

}

var employee1 = Employee()
employee1.name = "Tom"

var employee2 = employee1
employee2.name = "Fred"

print(employee1.name) // Tom
print(employee2.name) // Fred
```

In the second example, we declare a structure, `Employee`, and repeat the steps of the first example. The output of the print statements is different, though.

By replacing `class` with `struct`, the outcome of the example changes in a significant way. An instance of a class, an object, can have multiple owners. An instance of a struct has one owner. When an instance of a struct is assigned or passed to a function, its value is copied. A unique instance of the struct is passed instead of a reference to the instance of the struct.

The moment `employee1` is assigned to the `employee2` variable, a copy of `employee1` is made and assigned to `employee2`. The values of `employee1` and `employee2` have no relation to one another apart from the fact that they are copies.

Benefits of Value Types

Swift uses value types extensively. Strings, arrays, dictionaries, sets, and numbers are value types in Swift. That's no coincidence. If you understand the benefits of value types, you automatically understand why these types are defined as value types in Swift's standard library. What are some of the benefits of value types?

Storing Immutable Data

Value types are great for storing data. And they are even better suited for storing immutable data. Assume you have a struct that stores the balance of the user's bank account. If you pass the balance to a function, that function doesn't expect the balance to change while it's using it. By passing the balance as a value type, it receives a unique copy of the balance, regardless of where it came from.

Knowing that a value won't change is a powerful concept in software development. It's a bold promise and, if used correctly, one that value types can keep.

Manipulating Data

Value types usually don't manipulate the data they store. While it is fine and useful to provide an interface for performing computations on the data, it's the owner of the value that manipulates the data stored by the value.

This results in a control flow that is transparent and predictable. Value types are easy to work with and, another great benefit, they behave admirably in multithreaded environments.

Why is that? If you fetch data from a remote API on a background thread and pass the data, as a value, from the background thread to the main thread, a copy is made and sent off to the main thread. Modifying the original value on the background thread won't affect the value that's being used on the main thread. Clean, simple, and transparent.

When to Use Value Types

While I hope I’ve convinced you to start using value types more often, they’re not a good fit for every scenario. Value types are great for storing data. If you pass around the balance of someone’s account in your application, you shouldn’t be using a class instance. What you are doing is passing around a copy of the current balance of the account.

But if you need to pass around the account itself, then you want to make sure the objects that have a reference to the account are working with the same account, the same instance. If the name of the account holder changes, you want the other objects that have a reference to the account to know about it.

Value types and reference types each have their value and use. It would be unwise to choose one over the other. Andy Matuschak has a clear stance on the benefits of value types and reference types in software development and has given several presentations on the topic. Andy describes the objects of an application as a layer that operates on the value layer. That’s a great way to put it.

> Think of objects as a thin, imperative layer above the predictable, pure value layer. â€” Andy Matuschak

8 Catching Errors

I was thrilled to find out that error handling is built into the Swift language. As I wrote earlier, it's too easy to ignore errors in Objective-C. You can still ignore errors in Swift if you choose to go that route, but you need to be explicit about it.

This chapter won't repeat what The Swift Programming Language has to say about error handling in Swift. Instead, I'd like to take the opportunity to give you some tips about error handling. When should you handle errors? Should you always notify the user if something goes haywire? Where should you handle errors?

When Should You Handle Errors

The short answer is simple. Always. The answer is more nuanced, though. The remainder of this chapter zooms in on the when, the how, and the why.

Where Should You Handle Errors

Error handling in Swift is intuitive and flexible. You can propagate errors and handle them where it feels appropriate. The question is then, "Where is it appropriate to handle errors?" The rule I apply in most scenarios is simple and easy to adopt. Whenever an object performs an operation and an error is thrown, the error is handled by the object that still understands what the error is about. Let me explain what I mean by this with an example.

A few days ago, I was implementing receipt validation in one of my applications. I won't go into the finer details, but suffice to say that the application sends the receipt to a remote server, which validates the receipt, and returns a response to the application. The response is a JSON object that is deserialized using John Sundell's fantastic Unbox library.

As you can imagine, things can go wrong in several phases of the operation. Which objects should handle which errors? The object that deserializes the JSON response should handle any errors that are thrown by the deserialization operation. It isn't useful to propagate these errors since the objects upstream won't understand the errors and, more importantly, don't need to know about them.

If you do want to notify the objects upstream, then I suggest defining and throwing a custom error. This can be very useful to summarize a range of errors that an upstream object can understand and respond to.

Another example involves Core Data. I always use a dedicated object to manage the application's Core Data stack. This object handles any errors that are thrown by a save operation. If other objects need to know about a failed save operation, a custom error is thrown by the Core Data manager.

Objects that are unaware of the persistence layer, Core Data in this example, won't know how to handle the errors that a managed object context can throw. They can at best check if any errors are thrown by a managed object context. It usually doesn't make sense to make an object handle errors it doesn't understand or doesn't know how to respond to.

Notifying the User

Developers new to programming or Swift development usually choose one of two strategies when handling errors. The either ignore errors altogether or they print any errors that pop up to the console. Neither approach is ideal, but the latter group is at least informed when something goes wrong.

User Action

Many operations can go wrong or backfire. When should you notify the user? Let's start with the most obvious scenario. When the user performs an action that backfires, she should be notified that her action was unsuccessful.

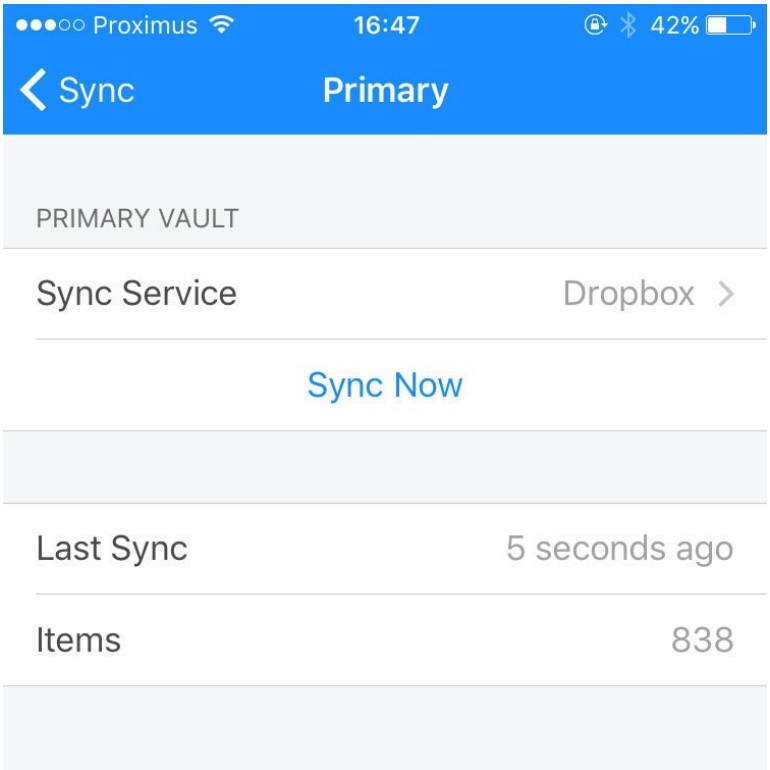
Is this a hard rule? No. If the user likes a post on Facebook and the API request was unsuccessful, you may only want to notify her if she's still on

the page she liked. If the failed API request is the result of a poor network connection, then that would mean the user is notified after the network request timed out. That may be too late for the user to make sense of the error.

Background Operations

Most applications perform background operations the user isn't, and shouldn't be, aware of. Examples include data synchronization and receipt validation. Should you notify the user if your application is unable to synchronize the user's database? The answer is almost always no. This doesn't mean the user shouldn't be informed, though.

What I like about 1Password's mobile application, for example, is the current state of the synchronization process as well as the last time the application synchronized.



1Password

If something goes funky, the user can read about it in the settings of the application.

Avoid Cryptic Messages

I'm sure you've seen error messages with a cryptic message or error code. You should avoid this at any cost. It doesn't help the user to see an alert with an HTTP status code. In fact, it may concern and annoy the user.

Monitoring Application Health

The downside of mobile applications is that you can't easily monitor your application's performance or health in real time. This is very different for web applications.

How are you going to stay informed about your application's performance and health the moment it's in the hands of your customers? Are you going to wait until your customers email you? We cover this in more detail later in the book. For now, I want to emphasize that you need eyes on your application from the moment your application is in production. How often does your application crash? Which errors occur most often? What is the cause of these errors?

Don't Ignore Them

I think that knowing how, when, and where to handle errors is what stops many inexperienced developers from handling errors properly. There is no clear-cut recipe for error handling and that can be frustrating. The guidelines I present in this chapter should give you a better idea where to start and I hope it's clear that you shouldn't ignore errors.

9 Using Fatal Errors to Write Elegant Swift

Speaking of errors, a few months ago I stumbled on a discussion in the thoughtbot guides about the use of fatal errors in Swift. It seems every developer has an opinion about fatal errors and a consensus hasn't been reached yet in the Swift community. The only mention in The Swift Programming Language is in the section that discusses the `guard` statement and early exit.

When I first encountered the `fatalError(_:file:line:)` function, it reminded me of the `abort()` function. Even though both functions cause the immediate termination of your application, the `fatalError(_:file:line:)` function is different and, in the context of Swift, it's more useful.

What to Do When You Don't Know What to Do

Ever since I started working with Swift, I've been struggling with the implementation of the `tableView(_:cellForRowAt:)` method. If you think that sounds silly, then take a look at the following example.

```
override func tableView(_ tableView: UITableView, cellForRowAt index\
Path: IndexPath) -> UITableViewCell {
    if let cell = tableView.dequeueReusableCell(withIdentifier: Sett\
ingsTableViewCell.reuseIdentifier, for: indexPath) as? SettingsTable\
ViewCell {
        // Configure Cell
        cell.textLabel?.text = "Some Setting"

        return cell

    } else {
        return UITableViewCell()
    }
}
```

There are several variations of the above implementation and I've tried all of them. The idea is simple. We expect an instance of the `SettingsTableViewCell` class if we ask the table view for a cell with the reuse identifier of the `SettingsTableViewCell` class. Because the `dequeueReusableCell(withIdentifier:for:)` method returns a

`UITableViewCell` instance, we need to cast the result to an instance of the `SettingsTableViewCell` class.

This is inconvenient since we always expect to receive a `SettingsTableViewCell` instance if we ask the table view for a cell with the reuse identifier of the `SettingsTableViewCell` class. We could use the `as!` operator instead of the `as?` operator, but that isn't a solution I feel comfortable with. Remember that I avoid the exclamation mark whenever I can.

If for some reason, something goes wrong, we return a `UITableViewCell` instance from the `tableView(_:cellForRowAt:)` method. But that should never happen. Right?

While this is fine and necessary to make sure we return a `UITableViewCell` instance from the `tableView(_:cellForRowAt:)` method, I hope you can see that we're implementing a workaround for a scenario we don't expect, a scenario that should never occur.

Guarding Against Unexpected Events

Every time I implement `tableView(_:cellForRowAt:)` I wonder if there's a better approach. And there is. We need to guard against the event that the table view hands us a `UITableViewCell` instance we don't expect and, if that happens, we throw a fatal error.

```
override func tableView(_ tableView: UITableView, cellForRowAt index\
Path: IndexPath) -> UITableViewCell {
    guard let cell = tableView.dequeueReusableCell(withIdentifier: S\
ettingsTableViewCell.reuseIdentifier, for: indexPath) as? SettingsTa\
bleViewCell else {
        fatalError("Unable to Dequeue Reusable Cell")
    }

    // Configure Cell
    ...

    return cell
}
```

The application crashes if a fatal error is thrown. Why is this better? This is a better solution for two reasons.

Unexpected State

If the application runs into a scenario in which a fatal error is thrown, we communicate to the runtime that the application is in a state it doesn't know how to handle.

In the first example, the *solution* was to return a `UITableViewCell` instance. Does that solve the problem? No. The application ignores the situation and avoids causing havoc by returning a `UITableViewCell` instance. The application avoids dealing with the unexpected state it's gotten itself into.

Finding and Fixing the Bug

If the application runs into a state we didn't anticipate, it means a bug has slipped into the codebase. If the application is terminated due to a fatal error being thrown, we have work to do. It means that we need to find the bug and fix it.

The above solution, using a `guard` statement and throwing a fatal error, is a solution I'm very happy with. It avoids the obsolete `if` statement of the first implementation of the `tableView(_:cellForRowAt:)` method and it correctly handles the situation. The result is a much more elegant implementation of the `tableView(_:cellForRowAt:)` method. And it adds clarity to the implementation of the `tableView(_:cellForRowAt:)` method.

Use Fatal Errors Sparingly

This doesn't mean that you need to use fatal errors whenever you want to avoid error handling or the application enters a state that's hard to recover from. I use fatal errors only when the application *can* enter in a state it wasn't designed for. Take a look at the following example.

```
 1 import Foundation
 2
 3 enum Section: Int {
 4
 5     case news
 6     case profile
 7     case settings
 8
 9     var title: String {
10         switch self {
11         case .news: return NSLocalizedString("section_news", comment\
12 : "news")
13         case .profile: return NSLocalizedString("section_profile", c\
14 omment: "profile")
15         case .settings: return NSLocalizedString("section_settings",\
```

```
 comment: "settings")
        }
    }

}

struct SettingsViewViewModel {

    func title(for section: Int) -> String {
        guard let section = Section(rawValue: section) else {
            fatalError("Unexpected Section")
        }
        return section.title
    }

}
```

The `title(for:)` method of the `SettingsViewViewModel` struct doesn't expect a value it can't use to instantiate a valid `Section` instance with. If the value of the `section` parameter *is* invalid, it wouldn't know what to do. In that scenario, a fatal error is thrown.

If the application *does* enter that scenario, it means you made a logical mistake and it's your task to find out why and how it can be resolved.

Clarity and Elegance

The use of the `fatalError(_:file:line:)` function has made my code more readable without compromising the inherent safety of the Swift language. It adds clarity to the code I write and Swift regains its elegance. Give it a try and let me know if you like it.

Part 3: Projects

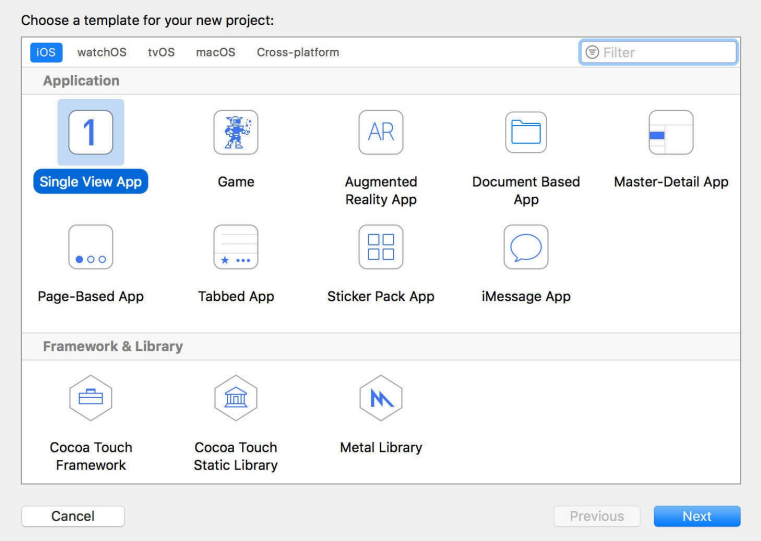
1 A Brand New Project

There are many tutorials and courses available to get started with Swift development. The issue with most of them is that they're focused on a specific topic, ignoring other essential aspects of Swift development. What teachers often forget is that developers new to a language, a framework, or a technology, are very, very receptive to new information. This includes best practices, but, unfortunately, it also means less good practices or even bad habits.

In this chapter, I'd like to walk you through the steps I take when setting up a brand new project in Xcode 9. By following the steps laid out in this chapter, you set yourself up for a successful project. Let it sink in and tweak it to your own preferences.

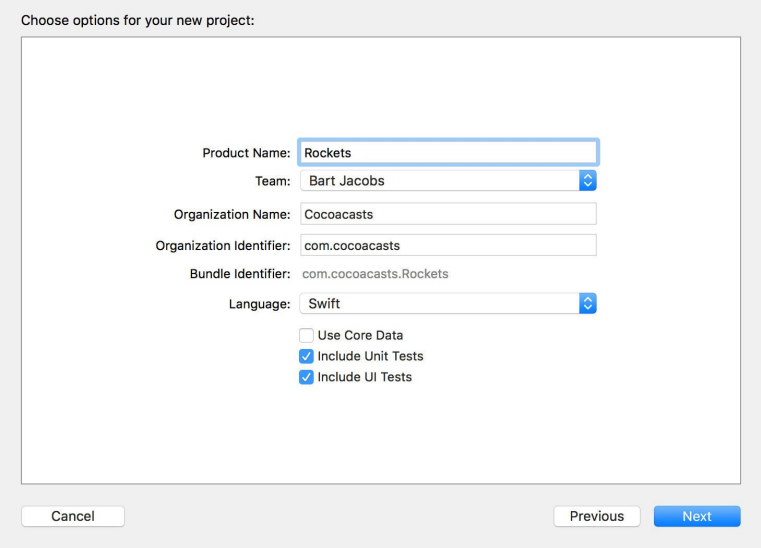
Step 1: Setting Up the Project

I understand that it can be useful to experiment from time to time, but I always use the **Single View App** template for new projects. It gives me an application delegate, a view controller, and a storyboard. I don't need anything else when I'm starting a new project.



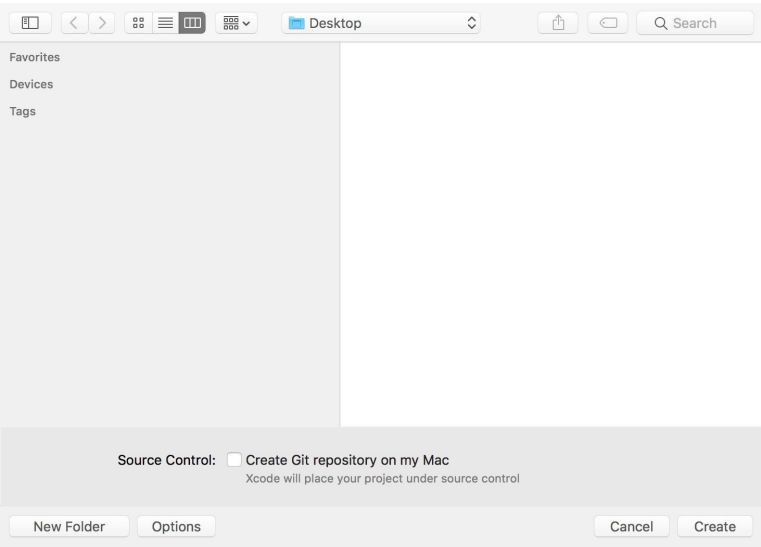
Choosing a Template

Give the project a sensible name and check **Include Unit Tests** and **Include UI Tests**. I never check **Use Core Data** for production projects.



Configuring the Project

While it may seem convenient to check **Create Git repository on my Mac**, I don't recommend this. Why is that? The project first needs a bit of housekeeping before I'm ready for my first commit, Xcode doesn't automatically add a **.gitignore** file, and I don't like Xcode's first commit message. The latter is obviously a nitpicky detail.

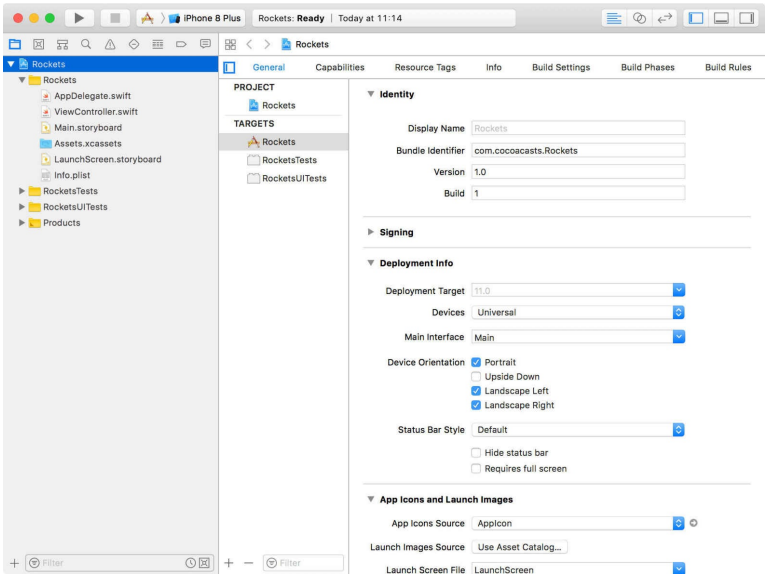


Choosing a Location for the Project

Step 2: Organizing the Project

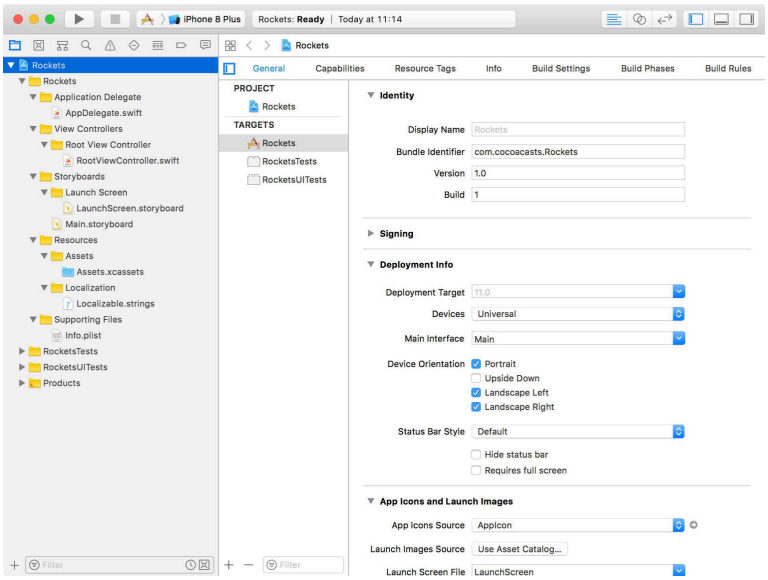
I always start by cleaning up the project. This simply means putting the application delegate in a separate group, organizing the view controllers, and making sure no files live at the root of the project. Small projects don't have the problem of becoming cluttered, but large projects do, and very quickly at that.

The idea is simple. This is the project Xcode has created for us.



Before Organizing the Project

And this is the project after adding some structure. Later in this book, I take a closer look at organizing a project in Xcode and how to use groups and folders to keep everything nice and tidy.



Step 3: Adding a README.md

Even if you're working in a team of one, it can be very helpful to document your project and it's something I strongly recommend. If you've ever browsed GitHub, then you probably know that putting a file named **README.md** at the root of the project is a good first step. Markdown is a wonderful markup language and it's ideal for this purpose. This is the README.md file I usually start with.

```
# Project Name
## Author: Bart Jacobs
### Description: Lorem ipsum dolor sit amet, consectetur adipiscing \
elit. Curabitur ac dolor justo, ac tempus leo. Etiam pulvinar eros a\
t lectus sollicitudin scelerisque. Aliquam erat volutpat. Suspendiss\
e eu eros non elit blandit suscipit. Morbi scelerisque euismod tempu\
s.

### Dependencies:
- Unbox
- RxSwift
- RxCocoa
- Reveal
```

If you're working in a team, then I recommend putting more time and effort into documenting the project. I discuss this in more detail elsewhere in the book.

Step 4: Build and Run

Before I put the project under source control, I make sure it builds and runs without warnings and errors. After updating the project's structure, it sometimes happens that Xcode can no longer find some of the files you moved. We need to fix this before we put the project under source control. As I mention in the chapter on source control, you should only make a commit if the project builds successfully, without warning and errors.

Step 5: Adding a .gitignore

Before we put the project under source control, we need to add a **.gitignore** file. If you're unfamiliar with Git and the purpose of a **.gitignore** file, then I recommend reading more about this fantastic version control system.

The **.gitignore** file at the root of the project defines which files need to be kept under source control and which files can be ignored hence the name of the file. This is what the **.gitignore** file of most of my projects looks like.

```
##########################
# **.gitignore** file for Xcode4 / OS X Source projects
#
# NB: if you are storing "built" products, this WILL NOT WORK,
#   and you should use a different **.gitignore** (or none at all)
# This file is for SOURCE projects, where there are many extra
#   files that we want to exclude
#
# For updates, see: http://stackoverflow.com/questions/49478/git-ign\
ore-file-for-xcode-projects
##########################

#####
# OS X temporary files that should never be committed

.DS_Store
*.swp
profile


####
# Xcode temporary files that should never be committed
#
# NB: NIB/XIB files still exist even on Storyboard projects, so we w\
ant this...

*~.nib


####
# Xcode build files -
#
# NB: slash on the end, so we only remove the FOLDER, not any files \
that were badly named "DerivedData"

DerivedData/

# NB: slash on the end, so we only remove the FOLDER, not any files \
that were badly named "build"

build/


#####
# Xcode private settings (window sizes, bookmarks, breakpoints, cust\
om executables, smart groups)
#
# This is complicated:
#
# SOMETIMES you need to put this file in version control.
# Apple designed it poorly - if you use "custom executables", they a\
re
#  saved in this file.
# 99% of projects do NOT use those, so they do NOT want to version c\
ontrol this file.
#  ..but if you're in the 1%, comment out the line "*.pbxuser"
```

```
57
58 *.pbxuser
59 *.mode1v3
60 *.mode2v3
61 *.perspectivev3
62 #    NB: also, whitelist the default ones, some projects need to use\
63  these
64 !default.pbxuser
65 !default.mode1v3
66 !default.mode2v3
67 !default.perspectivev3
68
69
70 ####
71 # Xcode 4 - semi-personal settings, often included in workspaces
72 #
73 # You can safely ignore the xcuserdata files - but do NOT ignore the\
74  files next to them
75 #
76
77 xcuserdata
78
79 ####
80 # XCode 4 workspaces - more detailed
81 #
82 # Workspaces are important! They are a core feature of Xcode - don't\
83  exclude them :)
84 #
85 # Workspace layout is quite spammy. For reference:
86 #
87 # (root)/
88 #   (project-name).xcodeproj/
89 #     project.pbxproj
90 #     project.xcworkspace/
91 #       contents.xcworkspacedata
92 #       xcuserdata/
93 #         (your name)/xcuserdatad/
94 #     xcuserdata/
95 #       (your name)/xcuserdatad/
96 #
97 #
98 #
99 # Xcode 4 workspaces - SHARED
100 #
101 # This is UNDOCUMENTED (google: "developer.apple.com xcshareddata" -\
102  0 results
103 # But if you're going to kill personal workspaces, at least keep the\
104  shared ones...
105 #
106 #
107 !xcshareddata
108
109 ####
110 # XCode 4 build-schemes
111 #
112 # PRIVATE ones are stored inside xcuserdata
113 !xcschemes
114
115 ####
116 # Xcode 4 - Deprecated classes
117 #
118 # Allegedly, if you manually "deprecate" your classes, they get move\
119 d here.
120 #
```

```
121 # We're using source-control, so this is a "feature" that we do not \
122 want!
123
124 *.moved-aside
125
126 # CocoaPods
127 /Pods
```

As you can see it includes references to Xcode 4, which shows you how long I've been using this **.gitignore** file. It works well for me. While most of the contents are easy to understand, there are a few details that are worth pointing out.

At the bottom, you can see a reference to CocoaPods, the dependency manager of my choice. The `/Pods` entry means that I ignore the entire **Pods** directory. This is a personal choice and it has its pros and cons. I've never run into the cons, which is why I've stuck with this option.

While ignoring the **Pods** directory is fine and up to you to decide, you should never ignore **Podfile.lock** or, if you're using Carthage, **Cartfile.lock**. As the extension of the file implies, a lock file locks the current configuration. A project's **Podfile** or **Cartfile** describes the project's dependencies, optionally specifying the version that should be used. The project's **Podfile.lock** or **Cartfile.lock** locks the current configuration, including the version that is used by the project.

Putting lock files under source control is especially important if you're working in a team. When a team member checks out the repository and installs the project's dependencies by running `pod install` or `carthage bootstrap`, you want to make sure they're including the correct version of each dependency.

The **.gitignore** file also defines that I ignore any personal preferences and settings. This is recommended for teams. You don't want to clutter the repository with the preferences of every team member that worked on the project.

To be clear, I didn't create this **.gitignore** file from scratch. I don't recall where I picked it up, but it's important to be critical of things you pick up from elsewhere. Make sure you understand every line of the **.gitignore** file you use in your projects.

Step 6: Putting the Project Under Source Control

The most popular version control system in the Cocoa and Swift communities is Git. It's a wonderful piece of software with many features and it's easy to get started with. This book won't cover the details of Git, but there are plenty of resources to get up to speed quickly. I reference a few of them in the chapter on source control.

When I wrote my first lines of code many moons ago, I didn't know about source control and discovering Git was a true revelation. It was one of those *aha* moments. It felt incredible.

It's surprising to see that there are still developers that don't use any type of source control. If that's you, don't be embarrassed. But take a moment to learn the basics of Git (or any other version control system) and take your work seriously. You don't want to lose any work by not using source control.

The steps I take to put the project under source control are always the same. Take a look at the following commands. That's all it takes to get started with Git.

```
1 git init
2 git add .
3 git commit -m "Project setup"
```

Step 7: Pushing the Project to GitHub

I very much enjoy using GitHub and, to make sure my work is safe, I create a private repository and push the project to GitHub. You can use other solutions as well, such as Bitbucket or GitLab. Once you've created the repository on GitHub, you only need to add it as a remote and push the repository.

```
1 git remote add origin <URL>
2 git push origin master
```

Step 8: Optional Steps

You can take it a few steps further and, if you're working in a company, that may be necessary. These steps usually include adding a **Gemfile**,

setting up continuous integration or creating a project in a management tool. These steps are optional, though.

2 Project Structure

The application templates that ship with Xcode aren't the best examples for developers new to Cocoa development. A new project in Xcode isn't structured in any way and the project folder on disk isn't either. This isn't an approach that scales and it's not something I can recommend.

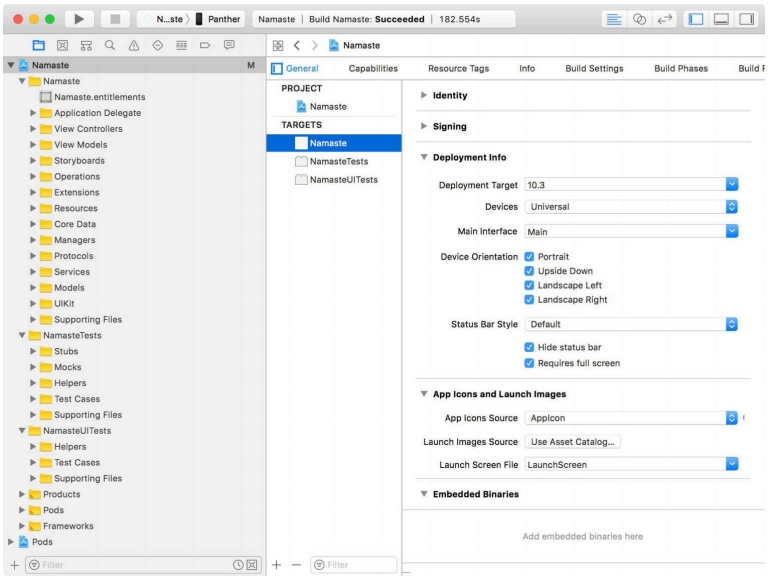
One of the subtle details students appreciate about my books and courses is that they see what a production project can look like. *Can* not *should*. I believe it's important to stick to a set of guidelines to make sure you introduce consistency and structure into your projects.

That's probably what I appreciated most when I created my first Ruby on Rails application. David Heinemeier Hansson emphasizes that Ruby on Rails is an opinionated framework providing sensible defaults. Such an approach has its pros and cons. A definite upside is that a Ruby on Rails application expects a particular project structure. Developers new to Ruby on Rails like this, but I can imagine that more experienced programmers develop their own opinions over time.

I'm sure you agree that any structure is better than no structure. The approach I use has evolved over the years. For example, when I started to adopt the Model-View-ViewModel pattern in my projects, I made a few tweaks to my projects' structure. That's fine. The core idea is to be consistent and to have a project structure that works for you and your team.

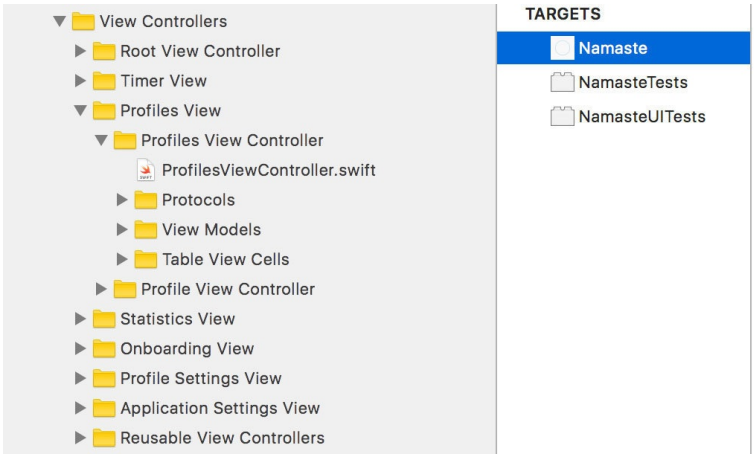
An Example

The projects I include in my courses and books are relatively basic compared to the production projects I work on. For that reason, I'd like to give you a peek behind the scenes of one of my recent projects. Every project I create and work on adopts this project structure. It works for me, but I understand that other solutions work equally well. Remember that consistency is key.



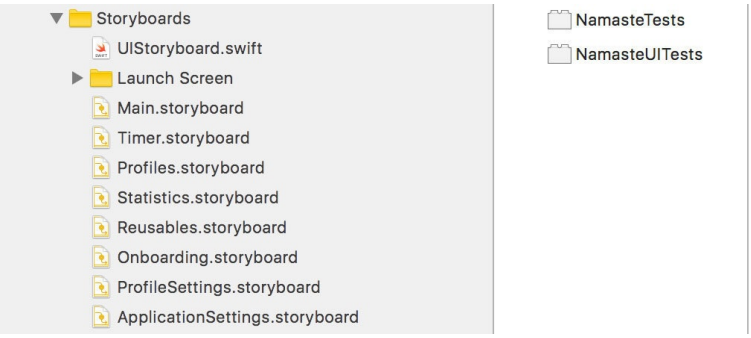
A project structure that works.

As I wrote in the previous chapter, the first thing I do when I create a new project in Xcode is refactor the project's structure. I put the application delegate in its own group and I create a group named, **View Controllers**, for the modules of the application. For me, a module is nothing more than a screenful of content or a reusable user interface.



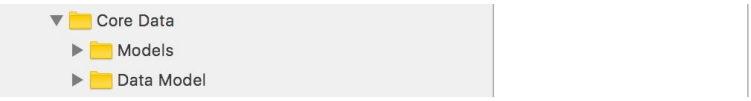
View Controllers

I'm a fan of storyboards and I put the storyboards of the project in a separate folder. I aggressively use storyboard references to make sure I don't end up with one large storyboard that's slow to load and difficult to manage.



Storyboards

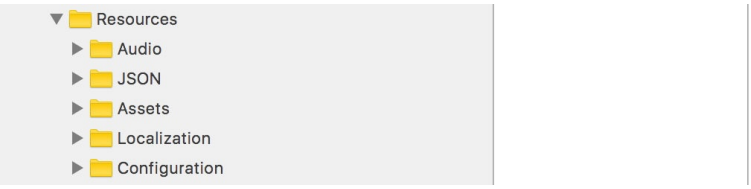
Everything related to Core Data is thrown in a separate group. This includes the data model and the extensions for the `NSManagedObject` subclasses.



Core Data

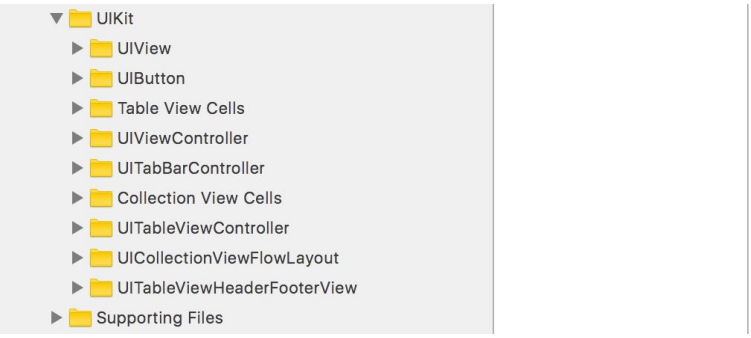
I also create a separate group for extensions, protocols, generic models, and managers. I try to avoid having groups that are named **Miscellaneous** or **General**. This doesn't help me organize my projects and it makes finding files inefficient.

A group named **Resources** contains the project's resources, such as assets, audio files, and localization files. Notice that I create subgroups to keep everything nice and tidy.



Resources

The group named **UIKit** contains subclasses of UIKit components. Everything has its place in my projects.



UIKit Subclasses

This project also uses the Model-View-ViewModel pattern and I have a separate group for generic view models, view models that are used throughout the project. The project also makes heavy use of operations. The `Operation` subclasses are located in a group named **Operations**.

The target's **Info.plist** file is located in a group named **Supporting Files**.

View Controllers

The **View Controllers** group is split up into several subgroups, one for each module of the application. This split is almost always specific to the project. Use your common sense and see what works best for your project.

Every module contains subgroups for the view controllers of that module and such a subgroup usually contains several subgroups, such as protocols, view models, table view cells, and collection view cells. This is very often specific to the project.

Test Target

The test target is organized a bit differently for obvious reasons. A group named **Supporting Files** contains the target's **Info.plist**. I create separate groups for the test cases, the extensions I use only in the unit tests, and I also create a group for stubs.

UIKit Subclasses

It frequently happens that I create mock and other helper types. These are also lumped together in separate groups, **Mocks** and **Helpers** respectively.

On Disk

Prior to Xcode 9, a group in Xcode's **Project Navigator** didn't correspond with a folder on disk. This has finally changed. In Xcode 9, a group by default corresponds to a folder on disk. This means that the project structure you create in Xcode more or less corresponds to a folder structure on disk.

If you're using an earlier version of Xcode, I strongly recommend that you manually create a folder for every group you create in Xcode's **Project Navigator**. This is tedious, but it's well worth the investment. And it isn't difficult.

You create a group in the **Project Navigator** and copy the name of the group. Open the **File Inspector** on the right, click the folder icon in the **Location** section, and create a new folder with the same name as the group in the **Finder** window that appears. Choose the newly created folder and click **Choose** to link the group to the folder. Any files or other groups that you add to the group in Xcode are automatically added to the folder that's linked to the group.

What Do I Gain

Sticking to a project structure makes it easier to find what I'm looking for, but it also allows me to focus on what matters, that is, the code that I write. Focus is a topic I mention a lot in this book because it's essential if

you want to get meaningful work done. An organized project that sticks to a structure that makes sense is one less distraction to worry about.

Tools

There are several tools and scripts that help you with organizing your projects. The best known tool is liftoff, created by the folks at thoughtbot. It's a command line tool that allows developers to quickly create a new Xcode project that sticks to a predefined project structure.

This can be useful if you're working in a team and every project needs to, and should, use the same project structure.

Caveats

This project structure works for many projects, but there are a few caveats. What happens if your project manager or the designer introduces major changes? In such a situation, you need to make changes to the structure of your project. That's the downside. The question is whether this is necessary and warranted. Shouldn't the developer start developing once the design and feature set is agreed on?

3 Project Hygiene

In the previous chapter, I wrote about structuring the projects you work on. This applies to every software project. What I didn't discuss in the previous chapter is project hygiene. What should be kept under source control? What do you do with old files and assets?

Projects Evolve

As a project grows and ages, more stuff makes its way into the repository. That isn't a problem because it's inevitable. Fortunately, the project is under source control. You're using source control. Right?

Whenever I inherit an existing project, I start by exploring the project's repository and its history. You'd be surprised by what I sometimes discover in projects. Two things that stand out, though, obsolete files and files that shouldn't be kept in a repository.

Obsolete Files

From the moment you no longer need or use a file, delete it. It's that simple. Some developers are reluctant to delete stuff and clean up a project. They usually have a clear motivation. It's either laziness or the fear that they might need it later. Some day.

Good programmers are lazy, but this isn't the type of laziness that the adage is referring to. This is sloppiness that you should avoid. If you're a team lead, make sure you avoid this type of behavior in your team. Project hygiene is just as important as taking your car to the shop every so often.

The fear some developers have when they need to delete a file is unwarranted if you're embracing source control. The idea underlying source control is exactly why you don't need to be afraid to experiment or delete stuff. Trust your version control system, make sure you have

backups of your repository, and hit the delete button when something's no longer needed.

Comments

The same applies to comments. Comments are great. I love comments, a bit too much according to some people. But comments shouldn't be used to comment something out, commit it, and save it for later. If a chunk of code is no longer used, then delete it. Should you need it later, then source control is there for you.

Whenever I see a piece of code commented out, a warning bell goes off in my head. Clean it up and commit it to the repository. Seriously. Embrace source control and learn to trust it.

Documentation

Keeping documentation up to date can be challenging. I'm not aware of a infallible solution to this problem. That said, make sure you regularly update the documentation of your projects. Take a look at the documentation and make sure recent changes are reflected in the documentation.

Did you switch from CocoaPods to Carthage? That means you probably need to update the project's **README**. Is there still a **Podfile** at the project's root? And what about the **Pods** project? Obsolete configuration files are more common than you think. Did someone experiment with a command line tool that generated a configuration file and didn't bother to removing it when they no longer needed it? It happens.

The best strategy that I know of to keep documentation up to date is surprisingly simple. It works best if it's made explicit by turning it into a policy or guideline. Whenever you make a change that contradicts the documentation, it's your task to update the documentation. If it's possible, update the documentation in the same commit that includes the code changes.

It's a healthy strategy for several reasons. First, it places the responsibility with the developer that made the change. That person is in

the best position to update the documentation. Second, it minimizes the risk that you forget to update the documentation.

Don’t Commit Everything

Keeping a project’s **.gitignore** file up to date is essential if you want to avoid cluttering your project’s repository. Files that are generated are generally ignored, such as builds and documentation. That’s also why I prefer to leave the **Pods** directory out of the repository. The project’s **Podfile.lock** and **Cartfile.lock** define the project’s dependencies, which s sufficient to install the dependencies and build the project.

Anything that’s related to the developer or the IDE you’re using should be ignored. Xcode keeps your preferences in the project folder and AppCode also stores a bunch of preferences at the project’s root. That’s fine, but you don’t need to commit those to the project. One of the reasons is that you don’t want to have conflicts with other team members.

Sensitive information should almost always be ignored. I once worked on a project that contained private keys. This isn’t something I recommend. Private keys should be stored in a secure location, not in the repository.

Guidelines and Exceptions

This chapter merely describes a set of guidelines. At some point, you will run into an exception. You need to discuss this with your team and what you decide may differ from project to project.

As I mentioned elsewhere in this book, be critical. It’s not because Bart Jacobs said that you shouldn’t keep the **Pods** directory under source control that that is what you should do. There are plenty of developers who disagree with this stance.

4 Document Everything

Like testing, documentation is often considered a luxury or “Something we can do later.” This is a common misconception and, especially in teams, it comes back to bite you.

There are various types of documentation. The one you’re probably most familiar with is the documentation or comments in the projects you work on. While it’s true that adding too many comments isn’t a good idea, I hope you agree that no comments is worse.

Start With the Basics

As a basic rule, every project should have a README at the root of the project, regardless of the project. Even if you’re creating a tiny library that adds rounded corners to a button, then you should add a README. Every time I come across a project on GitHub that doesn’t have a properly formatted README that says a few things about the project I’m looking at, I close the browser tab and move on.

You don’t need to spend days writing documentation. The basics are fine to get your feet wet. Start with a brief description of the project, its authors, its dependencies, and how someone unfamiliar with the project can quickly get up to speed. The latter is often overlooked. It happens all too often that I need to figure out which tools to install, which version of Ruby I need to have, and on and on only to compile the project. That shouldn’t be necessary and it can be avoided with minimal effort.

Project ownership, a topic I discuss elsewhere in the book, is often undervalued or ignored. The project owner is in charge of creating a step by step guide to help new developers get up to speed as quickly as possible. If you own the project, then it’s your responsibility to make sure such a guide exists and that it’s up to date. Did a new team member run into an issue? No problem. Debug the problem and update the README to make sure this doesn’t happen in the future. That’s not unreasonable. Is it?

Make It Easy

Documentation is only useful if people use it. If you hide the style guide of your team in a hard to find folder structure that's shared on Google Drive or Bitbucket, then the chances are that nobody's going to make an effort to look for it, let alone use it. It doesn't matter what solution you use, as long as you make sure your team can access it with a few clicks or key strokes.

There are several open source solutions that make it easy to automatically generate documentation for libraries or projects. This means that the documentation is automatically updated when you push a commit. Looking into an automated solution may be well worth your time.

You can even integrate your documentation with documentation browsers or make them available online. CocoaDocs is a nice example of this strategy. I believe it's no longer maintained, but the tools that power CocoaDocs are open source and available for you to use.

What to Document

The short answer is simple. Everything. The reason I recommend to document everything is simple. It's always a pain to figure something out and most of the time you don't have the time to mess around.

I remember that, years and years ago, I created a step by step guide to set up a LAMP stack on macOS. This changed with every release, and every time I installed a fresh copy of macOS or purchased a new computer, I had to figure out how to set up a LAMP stack. Having this step by step guide was a life saver. Every. Single. Time.

Up to Date

Keeping documentation up to date is challenging, especially if you work in an environment that evolves quickly. However, more frustrating than not having documentation is having incorrect documentation. I can't tell you how often I hit a wall because of incorrect or outdated documentation. You know it should work because it's documented. But it isn't working.

Keeping documentation up to date requires two elements. The first is the most important one, a mindset and commitment to do so. Again, ownership is key. Every page of documentation should have an owner. That person is responsible for keeping the documentation up to date. Avoid that one person is responsible for everything. This doesn't work.

The second element is having a robust system. This ties in neatly with discoverability. I've encountered situations in which an API was documented in several places. The date was my only indication which document was most up to date. That's what I assumed.

Some people swear by tools like JIRA and Confluence. That's fine and I'm not going to recommend a tool. I've never liked these tools. Confluence and me have never been best friends. Use a tool that you or your team enjoy using. It shouldn't take ten clicks and five browser windows to add a new entry to the documentation.

Make It a Core Tool

Documentation is often an afterthought. "Oh. We still need a system to manage the documentation. Confluence? Sure. Google Docs. That should work."

Don't make this mistake. Invest in a system, free or paid, that works for you and your team. If you find a free solution that works for you, then that's fantastic. If it's a paid solution, then that's fine too. As long as your team uses it on a daily basis, you're set and ahead of the curve.

I'm a Team of One

What do you do if you are a team of one? You're a freelancer or you run a small product business. The same rules apply. It simply means that you need to take care of everything yourself. The rules are simpler. You're the owner, which means that you're in charge of keeping everything up to date.

Don't make the mistake that you'll remember how to set up Jenkins six months from now. You won't. Write it down and document it. You'll be glad you did six months from now.

Some people go the extra mile and write a blog post. They document their frustrations by providing a step-by-step guide to, for example, set up a build server. Thousands of people will thank you for it.

Make It a Habit

If you're a team leader or a manager, then make sure your team makes documenting a habit. This can be challenging, but it's necessary. Every developer should learn that not every aspect of software development is fun or enjoyable.

A developer who makes documenting a habit understands what software development is about. It's more than writing code and making beautiful user interfaces. Software development is creating something that lasts. It should work today and a year from now. Documentation is a tiny gear in the machine that makes that possible.

Time Is Money

I can see why people sometimes see documenting as a waste of time since it has not measurable return on investment. Is that true? Are these the same people that claim that testing is a misuse of time?

Isn't it also true that no company measures how much time is lost figuring out things that should have been documented? Think about that and then consider the return on investment of proper documentation.

Part 4: Workflow

1 Testing

Developers often feel guilty when I start talking about testing. They sheepishly turn their head or mumble that they've been planning to write tests but haven't found the time. You shouldn't feel ashamed if you've never written a test. That said, you should consider starting today.

Where to Start

If you're unfamiliar with testing, then the subject feels abstract and complex. That's the feeling I had before I started writing unit tests. The truth is that testing isn't that difficult if you break it down. Like documentation, it's a topic many developers don't give much consideration.

Start Simple

The easiest tests to write are usually those that test the model layer of your application. If you're new to unit testing, then it's a good idea to start there. Don't start with testing view controllers or Core Data models. If you're new to unit testing, then that will only confuse you.

Choose a tiny class or struct, create a test target, and write your first unit test. Run the unit test and be surprised by how easy that was.

Code Coverage

While you can test a project without it, code coverage gives you an idea of how well your project is covered by your test suite. There are various tools and techniques for calculating code coverage. Apple added native support for code coverage in Xcode 7.

To enable code coverage, you check a checkbox in the configuration of your scheme and run the test suite. Xcode automatically collects coverage data, creates a report for you, and even displays code coverage metrics in the source editor.

Code coverage is merely an estimate. The implementation of Xcode, for example, inspects the code paths that are triggered when the test suite is run. Because Xcode spins up an instance of your application when it runs the project's test suite, code paths that are not unit tested are also triggered. Some of the application life cycle events in the application delegate, for example, are triggered when your test suite is run. I believe that Xcode takes these into account when calculating the project's code coverage. You can verify this in the test report Xcode generates for you.

Revealing Weaknesses

There's a more important reason why I was excited when Apple added support for code coverage to its IDE. Code coverage reveals weak spots in the unit tests that I write. Xcode shows you which code paths are triggered when the test suite is run. Green means that a code path is triggered by the test suite and red means a code path isn't triggered by the test suite. In other words, red means that I need to write more unit tests.

This occasionally happens when edge cases are in play. We have the tendency to test the happy paths and overlook or ignore the edge cases. What happens if the backend returns an empty result or the format of the file you're loading isn't what the application expects?

Writing Better Code

Even though writing unit tests isn't my favorite aspect of software development, over the years, I've come to enjoy and appreciate it. Not only is a robust test suite invaluable for any software project, it also makes you a better programmer. Why is that?

When you write a test for a function or method, you're forced to think about the implementation of that function or method. You approach the implementation from a different perspective.

Break It Down

Lengthy methods and massive view controllers are almost always symptoms of code smell, subtly telling you that it's time for a round of

refactoring. An important benefit of writing tests is that it pushes you to keep methods **short** and **focused**.

As a general rule, a method should focus on one thing and one thing only. If you adopt that strategy, methods no longer span dozens or, heaven forbid, hundreds of lines. Massive methods are very hard to test.

Not only is it difficult to understand what you're testing, too many variables are in play, each affecting the implementation and the corresponding tests. The number of code paths you need to test becomes unwieldy and the result is often that no unit tests are written or the method isn't properly covered by unit tests.

Keep It Simple

It's better to have a handful of short and focused methods than one monstrous one. Why is that? The most obvious benefit is that concise methods are easier to understand. It's easier to wrap your head around the method's implementation. You have a better understanding of what's going on and what the possible outcomes are.

Another important benefit is the naming of methods. If you create a method that spans dozens of lines and is responsible for half a dozen tasks, then what are you going to name that method? If you break that method down into smaller methods, each with a particular focus or task, naming these methods will be easy and straightforward.

By keeping it simple, you gain clarity. By creating massive methods or classes, you create chaos and lose focus.

Dependency Injection

From the moment I became serious about unit testing, I lost my fondness for singletons and came to like **dependency injection**. It felt as if I graduated as a programmer and no longer needed the singleton pattern to glue the pieces of a project together.

Dependency injection is a wonderful concept that's often ignored or discarded in favor of the singleton pattern. Even though I don't dislike

singletons, I always ask myself whether there's a better solution to solve the problem that doesn't involve a singleton. Hint. There often is.

Dependency injection, mocking, and testing is a powerful and flexible combination. Mocking and stubbing also become much easier with dependency injection.

Xcode and Testing

Xcode's support for testing has gradually improved over time. With its support for code coverage, you now get a nice test report and the source editor shows you if a particular code path was triggered by the project's test suite.

```
//
//  AccountViewViewModel.swift
//  KeepingItPrivate
//
//  Created by Bart Jacobs on 23/11/16.
//  Copyright © 2016 Cocoacasts. All rights reserved.
//

import Foundation

struct AccountViewViewModel {

    // MARK: - Properties

    let account: Account

    // MARK: - Public Interface

    var subscriptionAsString: String {
        switch account.subscription {
        case .monthly: return "Monthly Subscription"
        case .yearly: return "Yearly Subscription"
        case .trial: return "Trial Subscription"
        }
    }

    var expiresAtAsString: String {
        // Parse Date
        let date = parse(date: account.expiresAt)

        // Initialize Date Formatter
        let dateFormatter = DateFormatter()

        // Configure Date Formatter
        dateFormatter.dateFormat = "YYYY, MMMM"

        // Convert Date to String
        return dateFormatter.string(from: date)
    }

    // MARK: - Private Interface

    func parse(date dateAsString: String) -> Date {
        let dateFormat: String

        if dateAsString.contains("/") {
            dateFormat = "YYYY'/'MM'/'dd"
        } else {
            dateFormat = "YYYYMMdd"
        }

        // Initialize Date Formatter
        let dateFormatter = DateFormatter()

        // Configure Date Formatter
        dateFormatter.dateFormat = dateFormat

        if let date = dateFormatter.date(from: dateAsString) {
            print(date)
            return date
        } else {
            fatalError("Incompatible Date Format")
        }
    }

}
```

Code Coverage in Xcode

As I mentioned earlier, code coverage isn't magical and it has its flaws. You shouldn't solely rely on code coverage when writing unit tests. That

said, it makes it much easier to spot holes in your project's test suite.

My Current Test Setup

Even though XCTest has improved over the years, I usually use a few additional libraries for making testing easier and more powerful. For Objective-C projects, I have come to love and rely on OCMock, a mocking library for Objective-C. Because OCMock hooks into the Objective-C runtime, there currently isn't an equivalent for Swift projects.

OCMock allows you to test more aspects of your code. It's easy to mock objects and stub methods, enabling you to test parts of your codebase you wouldn't be able to test with only XCTest in your toolbox.

Another library that's great for testing is OHHTTPStubs. As the name implies, this library makes stubbing network requests effortlessly. With OHHTTPStubs, you no longer have an excuse to ignore networking logic in your project's test suite.

What Are You Testing

Not writing unit tests doesn't make you a bad programmer. But you should ask yourself *why* you're not writing unit tests. Most studies show that you save time in the long run because breaking changes are easier to find and bugs are less likely to creep into your project's codebase. It also gives you more confidence in the code you write.

Earlier in this chapter, I briefly touched on a topic that's very often overlooked by developers, especially those that are new to unit testing. What are you testing?

Implementation Versus Specification

From a developerâ€™s perspective, unit testing an entity means writing unit tests for every method and property of the entity. In other words, you're writing unit tests to test the implementation of the entity.

What's often overlooked or misunderstood is that unit testing falls in the category of black-box testing. This means that you don't care how the entity under test does what it does. From the moment you start writing

unit tests, you need to stop being a developer and take on the mindset of a tester.

As a tester, you're not interested in how something works. You carefully inspect the public interface of the entity and make sure the entity behaves as specified. You can compare this with someone testing a car. If the tester starts the car, she's not interested in what happens under the hood. The specification says that turning the key should start the car within a predefined period of time. The tester tests the behavior and functionality of the entity under test.

Public Only

By embracing black-box testing, unit testing becomes much easier. You donâ€™t care about the private methods and properties the entity defines. As a tester, you only focus on the public interface of the entity. By unit testing the public interface of the entity, you automatically unit test the private interface. That's the underlying idea.

Code Coverage Can Help

There is one caveat you need to be aware of. As a tester, you're not interested in the private methods and properties of the entity you're testing. That's fine because that's what black-box testing is about. But it also means that you need to carefully craft the unit tests you write. It can often mean that you need to write multiple unit tests for one public method or property. Why is that?

If you want to have complete code coverage for the entity you're unit testing, you need to make sure every code path of the entity under test is executed, including those of private methods and properties.

As I discussed earlier in this chapter, Xcode can help you with this. If you enable code coverage, Xcode to collect code coverage data for you. It visualizes code coverage for an entity in the gutter on the right of the source editor. The number indicates how many times the unit tests entered a particular code path. This is helpful to find out where your unit tests fall short.

It Takes Time

When I first started unit testing Cocoa applications, I tried to reach complete code coverage by unit testing public and private methods. In Objective-C, this is possible with a few tricks. Unit testing private methods and properties isn't possible in Swift. And that may be a good thing as I explained earlier.

From the moment you start to write unit tests for a project, you need to step out of your role as a developer and take on the mindset of a tester. You need to view the entity you're unit testing as a black box that needs to conform to a specification. It's the specification you need to test, not the implementation.

2 Continuous Integration

While continuous integration is especially useful for teams, I believe it also has benefits for developers that work solo. The practices I describe in this chapter apply to anyone developing software, large of small.

What Is It

The name continuous integration may sound daunting or scary, but it really shouldn't. The idea is simple. One or more developers work on a project and push their changes to a shared repository. Whenever a push is detected, a series of automated steps is set into motion. This usually involves creating a build, executing a test suite, and performing a number of quality assurance steps.

There are several benefits to using continuous integration. One of the benefits I appreciate most is that no human interacts with the final product after code is committed to the repository. Other benefits include test automation, code analysis, and build optimization.

Why Is This Useful

Continuous integration is useful for most types of software development. There are many flavors of continuous integration available. The benefits of continuous integration are most apparent in teams with multiple developers working on the same codebase and committing multiple times a day.

Instead of working on a feature in isolation for days or weeks, you frequently commit and push your changes to a shared repository. That's when the build server jumps into action, building your application.

The reason I want to include a chapter on continuous integration in this book is to illustrate a common set of problems that I have seen in some of the companies I worked for. I won't be covering continuous integration

itself in great detail because there are many solutions available, each with their own benefits, paid, hosted, and open source.

Long Hanging Fruit

For small, personal projects, I don't recommend using a continuous integration solution because it comes with a bit of overhead and cost. For critical projects, though, continuous integration brings clear benefits to the table. What you define as critical is up to you. For me, any project for a third party is critical, large or small.

If you or your team are new to continuous integration, then I suggest to start with a simple setup to take advantage of the immediate benefits continuous integration has to offer. You may want to start with a hosted solution if you're new to continuous integration. There are open source solutions, but it can take some time to set up a build server, configure it, and integrate it with services, such as Apple's TestFlight or Google's Fabric.

To get your feet wet with continuous integration, try out a hosted solution first. If you like the benefits it brings, then consider a better solution, with better meaning more robust, scalable, and cost-effective.

The minimum setup I recommend is surprisingly basic. It includes automated unit tests with every push, updating of translations, and creating a build for testing. Is that enough? Is that worth it? I believe it is. It's too easy to forget to run unit tests before creating a test build. Right? It's certainly too easy to update the project's translations before creating a build, production or staging. And messing with test builds is something you want to avoid altogether.

Give it a try. Spend a few hours or a day setting up continuous integration for a project and evaluate the benefits you and your team get from it.

Avoid Human Tinkering

In an ideal scenario, you want to make sure that no human touches the product from the moment the last commit is pushed to the shared repository. This may sound obvious to some people, but it certainly isn't what happens when things hit the fan. Manually importing translations,

tweaking build flags and settings, updating version and build numbers, … These are steps that should be avoided at any cost. If you're scared to push a build to production, then you have work to do. Continuous integration can help you with this.

Let me repeat what I wrote earlier, you need to prevent that a human interacts with the final product after the team committed their last changes to the repository. The last thing the developer does is pushing their changes to the shared repository.

TestFlight

I used to use Fabric for distributing test builds to testers. Apple's acquisition of TestFlight has changed this for me. The reason is simple. I only need to send one binary to Apple's servers. The same binary is used by Apple to create a test build and a build for the App Store. My job is limited to selecting the build I want to deploy to the App Store. What your testers are testing is what your customers are going to download and use. Fantastic.

This is the goal you need to try to reach regardless of the solution you're using. The build that your testers are testing should be identical from a functional perspective to the one you send to Apple. This implies that both builds reference the same commit in the repository. TestFlight makes this very easy. Other solutions, such as Fabric and HockeyApp, may require you to do a bit of additional work since these solutions don't neatly integrate with iTunes Connect as far as I know.

Choose a continuous integration solution that integrates with TestFlight, if you opt for TestFlight, and that's able to send builds to iTunes Connect. You don't want to do this manually. Remember that choosing the correct build or archive to upload to iTunes Connect is another manual step you need to avoid. You don't want to be the one that uploaded the wrong build to iTunes Connect.

Painless Releases

You need to avoid that releases are stressful and chaotic. An automated workflow can and should help you with this. I realize that not every developer has the fortune to work in a company that embraces

automation and testing. Why don’t you make a start by writing a build script that automatically updates the build number?

You can even leverage existing open source tools for that, such as fastlane, a powerful suite of command line tools that automate common tasks, such as building, testing, generating screenshots, and sending builds to iTunes Connect.

Make It Robust

In the chapter on automation, I talk more about the benefits of automation. It’s important that you make your automation robust. Make sure it’s tested and doesn’t break several times a day. This isn’t only frustrating to anyone involved, it also makes that people don’t put a lot of trust into the solution. When people don’t trust a system or a solution, they tend to avoid it. That’s something you need to prevent.

If you’re working in a company that has the resources, then it pays off to put someone in charge of continuous integration, making sure it’s monitored and kept in check. In a small team, it’s usually a developer that’s in charge of automation and continuous integration. This is unavoidable if you’re small, but it can lead to frustration and a system that simply doesn’t work. Don’t consider automation and continuous integration an afterthought. It can save you time and money. In professional environments, it’s a core element of the workflow and release cycle.

Transparency

Transparency is an important aspect of working in a team. It means that people are allowed to make mistakes, but it also means that everyone involved is notified when something goes wrong.

Notifications

If a build fails or unit tests don’t pass, then every person working on the project, including the project manager, should be notified of this event. It should always be obvious who needs to take action to fix the problem.

If the team's workflow is sloppy and not respected by the people working on the project, then notifications pile up and they're ignored. This is very similar to warnings in Xcode. From the moment you ignore one warning, it's very easy to ignore the second and third warning. After a while, you no longer pay attention to warnings, which is a critical, and often painful, mistake.

Notificationitis

As I already mentioned, carefully tweak the notifications you send and receive. From the moment people receive too many notifications, they start to ignore them. That's one of the reasons I'm a big proponent of Inbox Zero. If your inbox is empty, you're less likely to ignore or miss important notifications.

Learning Curve

There is a cost to every continuous integration solution. There's a financial cost and there's a learning curve. You can opt for an open source solution, such as Jenkins, but that means you're responsible for setting up the installation as well as its maintenance. Hosted solutions are easier to get started with but you pay a price for them. This shouldn't be a problem for medium to large companies, but it can be for freelancers or independent developers.

I can assure you, though, that the investment you make pays itself back in the long run. The development team can focus on what they do best, quality improves, and releases are less stressful.

3 Refactoring

Refactoring is a word I frequently use in my commit messages because it's something I do very often. It's a fancy word for writing or implementing something differently. You don't often hear or read about refactoring. Especially if you work in a company or organization that puts an emphasis on output or values quantity over quality, there's little room or understanding for refactoring.

I have a different opinion on refactoring. Refactoring is an integral part of my workflow and there are several reasons why that is.

Technical Debt

Whenever I implement a solution to a complex problem, the primary goal is to get the solution to work. I need to know that the basic implementation I've put into place works as expected. This implementation is rarely the one I want to keep, though. There are several reasons for that.

I don't know about you, but I know myself well enough that I need several shots at a complex problem before I come up with a solution I'm happy with. It's not so much a matter of liking as it is avoiding technical debt. Refactoring is an often overlooked cure to technical debt. To be honest, it's not a cure, it's a vaccine.

Your first shot at an implementation isn't going to be your best. If you think about it, it's odd that other phases of a project get multiple opportunities to come up with a solution. Designers are often given the freedom to try out several designs. A developer is rarely given this luxury.

One of the reasons is that the client or project manager can see the design. Your implementation is usually not put under a lot of scrutiny if it works as expected. But, if you have a bit of experience developing software, you know that there's *working* and *working*. If a developer tells

me “It works.” and he shrugs his shoulders, then I know it’s time for a code review.

Core Data is a fine example. If you’ve worked with Core Data, then you know that getting the data model right is a crucial aspect. It’s not unusual that a data model changes over time as the application gains features and complexity. Those changes, however, are evolutionary. They build on what you already have. Getting the data model wrong in a fundamental way, however, is a pain. You don’t want to go through several lightweight or heavyweight migrations to fix something you could have fixed from the start.

I’m currently working on a project with a complex timer engine. It has taken me several rounds of refactoring before I ended up with a solution I’m happy with. Why is that? Implementing a solution to a complex problem often feels like putting together a puzzle in the dark. It takes time before you see the bigger picture. It takes time before you can see every component that’s involved and the various edge cases.

Building for the Future

While refactoring is a useful tool to reduce technical debt, there are several other benefits. Instead of tackling the problems you’re currently facing, you can think ahead. What’s the direction the product is going in? What problems does the product solve three or six months from now? Can we, with minimal effort, lay the groundwork for a solution that solves these problems?

There’s a subtle difference between premature optimization and simply optimizing for growth or new features. Let’s take Core Data as an example. I already mentioned that getting the data model right is very important. You want to touch the data model as little as possible. The cardinality of a relationship, however, is something you can be flexible with early on. It’s possible that an account only has one user at the moment, but is it plausible that the product supports multiple users per account in the not so distant future? Yes?

Think ahead and define in the data model that an account can have multiple users. And to make your current work easier, define a computed property, `user`, that fetches the first user. In other words, you’re currently

working in code as if an account has one user, but the data model already has support for multiple users.

The line between premature optimization and anticipating growth is fine, but it really pays off to spend some time considering these decisions. Code reviews and analysis can really help with this. Let yourself be challenged by another team member. Listen to their arguments, questions, and input.

The Fallacy of Sunk Cost

It's possible that you've invested hours, days, or even weeks into the implementation of a solution to a challenging problem you're faced with. You've poured hours and hours into this solution. It works, well, most of the time. The longer you spend on the problem and implementing a solution, the more you realize that your solution is flawed. That can be a hard pill to swallow.

Some of us are too proud to throw their work into the trash and start anew. Sometimes you don't even have that luxury. It can be particularly frustrating if you're working for yourself and you've invested time and money into a solution that you're not happy with.

It takes courage and confidence to take responsibility and admit to yourself that you need to come up with a new solution. Many of us are blinded by the fallacy of sunk cost. You've invested time and money into a problem and that means it needs to make its money back. The truth is that you won't ever get that time or money back. However, you can save yourself time and money in the future by taking another look at the problem, learning from your mistakes, and coming up with a better solution.

If you think this only happens to rookie developers, then you'd be mistaken. It's the experienced developer that has the audacity to start afresh with a clean slate and that knows and sees the solution won't cut it, for whatever reason. But, again, there's a fine line between knowing when the solution isn't good enough and striving for a perfect solution that doesn't exist. Be honest with yourself and, as always, ask advice from team members or fellow developers.

Starting Anew

Starting with a clean slate is almost always a hard sell if the client makes the decisions. Unfortunately, I've worked on projects that would have cost less if the client had had the courage to start anew. That's the fallacy of sunk cost. Truth be told, it's difficult to predict the future and it's easy to write this in hindsight.

While refactoring is an integral aspect of software development, you sometimes need to have the audacity to start from scratch. Learn from the mistakes that were made and create something that is ready for the future. While it's rare that you're given this opportunity, you can erase months or years of technical debt that slow the product's evolution down by starting from square one. That's a very tempting offer for a developer. Is it not?

4 Source Control

The day I discovered source control was one of those *aha* moments. I'm a self-taught programmer and that was clearly visible in the early days of my career. I'm sure you can guess what type of source control strategy I used before I found out about Git. Let me give you a hint. It involved copying folders. Ignorance is bliss, but not if you're a developer.

Whenever I'm hooked by a new tool or technique, I try to learn everything there is to know about it. Git was no different. In this chapter, I show you what you absolutely need to know about source control, the basics. I take it one step further by introducing you to the workflow that I've been using for the past few years, a workflow that's been working very well for me.

The Basics

The most popular source control solution in the Cocoa and Swift communities is Git. That doesn't mean it's the best solution, but it's the one I have the most experience with. I love it. If you're new to source control, then I urge you to put down the book and learn more about Git. Seriously. Put the book down. I'll be here when you come back. The folks at Fournova have a bunch of great resources to learn the basics.

It's nice to see that Xcode has improved support for Git over the years. Xcode 9 takes it up a notch and it sure looks promising. I use a combination of the command line and a dedicated Git client, Tower.

Don't Break These Rules

Elsewhere in the book, I write that once you know the rules and what they stand for, it's fine to break them. That rule does *not* apply to source control. There are a few simple rules I stick to religiously. Let me give you an overview.

Master and Develop

A project should always have a minimum of two branches, `master` and `develop`. When you create a new Git repository, you get `master` for free. Do yourself a favor, create a `develop` branch, and check it out before you start development.

Master Is the Truth

Deployments to production always happen from `master`. There should be no exceptions to this rule. Plain and simple.

Develop on Develop

Development should always happen on `develop` or a feature branch that branches off of `develop`. There's one exception to this rule, hotfixes.

Whenever you need to fix an urgent problem in production, you create a hotfix branch that branches off of `master`. You implement the solution on the hotfix branch and merge the changes back to `master`, ready to be deployed.

Technically speaking, you don't develop on a hotfix branch. You only fix the problem at hand.

Commits

Commits and commit messages are your window into the project's history. This means that it's important to carefully craft your commits and commit messages. That's one of the reasons I use a Git client because it allows me to pick and choose which code changes make it into the next commit. You can do the same from the command line, but it's less convenient. How you interact with Git is a personal choice.

There are several methods for creating a commit message. It doesn't matter which one you choose, but make sure you're consistent and, if you work on a team, adopt the same methodology across the team.

Avoid large commits. It's true that this can sometimes be a problem if you're modifying the project file of an Xcode project. Not only is this important for clarity and communication, but it's also important for practical reasons. If you want to cherry pick a commit, then you want to

make sure the commit you're cherry picking only includes what you think it includes. The same is true for rolling back commits.

Stashing

I'm always surprised by the number of developers that are new to the `git stash` command. The concept is surprisingly simple. Imagine that you're working on a new feature on a feature branch. Your project manager taps you on the shoulder and asks you to hotfix a problem on `master`. Your workspace is littered with changes. What do you do? Do you commit what you're working on just to make sure you don't lose your work? No. Don't do that.

The `git stash` command stashes the changes in the working directory and the staging area for later use, leaving you with a clean working directory and staging area. The stashed changes are pushed onto a stack and you can create as many stashes as you need.

To stash the changes in the working directory and the staging area, you execute the `git stash` command. The changes in the working directory and the staging area are safely stored for later use. You switch to the release branch, create a build, return to the branch you were working on, and apply the stash you created earlier. It's that simple. To apply a stash, you pop it from the stack of stashes with the `git stash pop` command.

If you frequently use stashes, I recommend naming the stashes you create. This makes it easy to find stashes and apply or remove a stash by name.

Patching

In a way, patching is similar to stashing. Both commands group a number of changes that you can apply later. A patch combines several commits or the changes in the working directory. The changes are stored in a file with a **.diff** extension.

Patches are useful for several reasons. It's easy to store patches to disk or share them with other developers. You can apply patches to branches no matter what the branching model looks like. They're also a good alternative to cherry picking.

As I mentioned earlier, I interact with Git using a combination of the command line and Fournova's Tower. Creating a patch is very easy in Tower. Select the commits you want to include in the patch, right-click, and choose **Save Patch for X Revisions…** from the contextual menu.

Patches can be used in a wide range of scenarios. Imagine you're working with a teammate on a big, complex feature. You're both working on the same branch and firing on all cylinders, committing as if your life depended on it. You're about to push your changes to GitHub when you notice that your teammate already pushed her changes. It's time to pull and perform a nasty merge. I don't know about you, but I don't like these kinds of merges.

The solution is surprisingly simple. You create a patch, include the commits you're ahead of the remote branch, and reset your branch to `HEAD` of the remote branch. You then pull the changes of your teammate and apply the patch you created earlier. If you're both working on the same branch and modifying the same files, then it's still possible that you still need to perform a manual merge.

Keep in mind that a reset is a potentially dangerous operation that can lead to data loss. Always bear this in mind when using the `reset` command.

Git Flow

A project with any degree of complexity needs a robust branching model. The branching model I have come to appreciate is the one outlined by Vincent Driessen. Vincent's model may look daunting at first, but it's easy to adopt in most software projects.

The idea is simple. A project adopting Vincent's branching model has a `master` and a `develop` branch. We covered that earlier in this chapter. If you decide to work on a feature, you create a feature branch that branches off of develop. When a feature is ready to be released, the feature branch is merged back into `develop`.

When the time comes to schedule a release, a release branch is created that branches off of develop. The goal of the release branch is to prepare the release and fix critical bugs. A release branch is never used to

implement features. After a successful release, the release branch is merged into `master`, merged into `develop`, and merged into downstream release branches.

Merging into `develop` and downstream release branches is important to guarantee that any work done on the release branch, such as bug fixes, is also included in feature and downstream release branches.

Earlier in this chapter, I discussed hotfix branches. If you need to push a hotfix to production, you create a hotfix branch that branches off of `master`. After releasing the hotfix, the same merge strategy is applied as for release branches.

This branching model works really well for me and the projects I work on. I recommend prefixing branch names with **feature/**, **release/**, and **hotfix/** to avoid confusion. If you apply this naming convention, then most Git clients automatically group every branch type in a folder.

Some Tips

I'd like to end this chapter with a few tips that I picked up over the years.

Does It Build

You should only make a commit if the project builds successfully, without warning and errors. This isn't always easy, especially if you're refactoring a section of a project.

Ignore What You Don't Need

Make sure you use an up to date **.gitignore** file. Elsewhere in this book, I share the **.gitignore** file I use in Cocoa projects. Regardless of which **.gitignore** file you use, make sure you understand what it includes. Don't include one you found on the web if you don't understand what it ignores.

Configuration

It's important to know and understand which files to keep under source control. A developer checking out the project should be able to build and run the project with minimal effort. This means that you don't need to

include the **Pods** directory if you’re using CocoaPods, but you need to include the **Podfile** and **Podfile.lock** files.

Sensitive Information

Sensitive information, such as private keys, should not be kept under source control. Make sure you don’t compromise security by dumping everything in the project’s repository.

5 Dependencies

The number of open source projects grows every day and it's pretty amazing what you can build with the help of open source software. Swift, for example, is an open source project that gained a lot of momentum the moment it was open sourced.

But open source projects also have a downside and so does every third party project you rely on. Do you remember Parse, the company Facebook acquired several years ago. A few years after the acquisition, Facebook announced it was going to shut down the company, leaving tens of thousands of developers, companies, and projects in the cold. The winners of this decision were freelancers and consultants. I also had to migrate a project away from Parse.

Why am I telling you this? It's true that open source projects and third party solutions can significantly speed up the development of a project. In very little time, you can create a proof of concept or a minimum viable product. This speed comes at a cost, though. As I mentioned earlier in this book, there are very few shortcuts in software development. Every shortcut you encounter has another side, a risk.

Minimize Dependencies

As a freelancer, I have the opportunity to browse many codebases. Some of them are great. Many of them aren't. One of the most common problems is that projects include too many dependencies. There's nothing wrong with having dependencies. Every modern software project has dependencies. The problem is that they're often used as a shortcut or because they're considered indispensable.

Whenever I create a project in Xcode, I start out without a dependency manager. That used to be different, though. Several years ago, after discovering CocoaPods, I immediately installed this Ruby gem on my machine. Don't get me wrong. CocoaPods and Carthage are fantastic

projects that have helped me tremendously. But that doesn't mean that you can't write a proper Cocoa application without them.

What Is a Dependency

Dependencies come in many forms and shapes. The most obvious dependencies are open source solutions you manage with a dependency manager like Carthage, CocoaPods, or the Swift Package Manager. Managing dependencies with a robust dependency manager is trivial. It's almost too easy.

Inexperienced developers make the mistake of including too many dependencies. After a while, the project feels like a car held together with tape and glue, ready to crash and burn at the slightest bump in the road. It's true that dependencies can significantly speed up the development of a project, but, again, shortcuts come at a price.

Remember that Parse offered developers a tempting solution, a mobile backend with support for data storage, push notifications, and more. Most of the projects that relied on Parse were forced to pay a hefty price when the platform was shut down.

As I mentioned earlier, dependencies come in many forms and shapes. Swift is a dependency of every project you work on. If you're a Swift developer, then you have no other option. Some dependencies are unavoidable. Xcode is probably another dependency of your projects. That too is out of your control unless you switch to JetBrain's AppCode.

RxSwift may also be a dependency of your project. You could roll your own library for reactive programming. Should you? I answer that question in a moment.

And don't forget the hidden dependencies of a project. If your project depends on Moya, then it automatically has a dependency on Alamofire.

Fewer Dependencies

The idea is simple. The fewer dependencies your project has the better. I hope we can agree on that. Every time I add a dependency to a project, I carefully analyze whether it deserves to be part of the project. What are

the alternatives? How long would it take me to implement a custom solution? How healthy is the dependency? Is it a liability for the project?

Every audit I set up with a developer or a company starts with a session that, among other things, analyzes the dependencies of the project. The lead developer of the project must be able to justify why the project depends on a particular library, platform, or technology. If the project uses CocoaPods to manage dependencies, for example, we inspect the contents of the project's Podfile.

I understand that dependencies can save time and, in the short term, money. But what happens if you need to replace a dependency that's used throughout the project? What if you need to switch out a vital component of the project and replace it with an alternative?

Mapping the Liabilities of a Project

Every dependency is a liability. Plain and simple. Even the Swift programming language is a liability. Don't believe me? Ask the tens of thousands of developers that adopted Swift in its early days. Migrating projects to the latest Swift syntax was no easy task. I can assure you.

Every project has points of failure. That's unavoidable. You, as a developer, have the responsibility to know about the points of failures of the projects you work on and, equally important, to minimize their number.

Don't Make Your Life Too Easy

I have a few rules I religiously stick to when it comes to dependencies. Every dependency I add needs to earn its way onto the project. The first rule I never break is simple. I directly interact with first party libraries and frameworks. What does that mean?

Core Data is the example I usually bring up in this context. I never use a Core Data library to interact with the framework. Core Data's API has improved substantially over the years and it's easy to use and understand. Why would you use a third party library to interact with the framework? Is it because you don't quite understand how the framework works? Convenience? Laziness? Be honest with yourself.

Most of Apple's frameworks are very well designed. The company has decades of experience designing APIs. It's true that some of the older APIs aren't pretty and a bit verbose, but don't let that be a reason to add another dependency to a project. Write your own lightweight wrapper if you want convenience and elegance. I guarantee you that it's worth the investment.

Rolling Your Own

Projects with dozens of dependencies very often have several dependencies for trivial tasks, such as a helper library for Auto Layout, a Core Data library, a convenience library for animations, and on and on. Whenever you're about to add a dependency to a project, consider how long it would take you to roll your own implementation. Would it be a matter of hours or weeks? If it only requires a few hours of work, then it may be better to roll your own. You probably don't need every feature of the dependency anyway. Is it feasible to create a lightweight solution that does the job equally well?

This doesn't mean that you always need to reinvent the wheel, but your first reaction should not be looking for a dependency to solve an issue. You're a developer. Right?

Choose Wisely

I already wrote that a typical software project has many hidden dependencies, such as the programming language and the developer tools we use. Some of these dependencies are easily overlooked. How do you collect crash reports or analyze user engagement? I bet you don't use Apple's analytics. Do you rely on Fabric, Firebase, or Mixpanel? These are mature platforms, but they're dependencies too.

Becoming a Better Developer

Relying on dependencies won't make you a better developer. In fact, you risk becoming lazy and complacent. Have you heard that a good developer is a lazy developer? This is not the type of laziness that's meant by this.

I'm a big fan of open source software. Unfortunately, the popularity of open source software has a dark side. Projects like Alamofire and Moya are fantastic. A committed group of developers has made it their goal to create a beautiful piece of software for everyone else to use. Projects like Alamofire are incredibly popular. They make your life as a developer easier.

The downside is that many developers have come to rely on Alamofire without learning the basics first. A surprising number of developers doesn't know how to use `URLSession` to perform a network request. Is that a problem? I believe it is. If I were to interview a developer for a position and he or she doesn't know how to perform a network request, I wouldn't hire that developer.

Samsara is an application I have been developing for several years. It makes a handful of network requests. In such a scenario, a networking library like Alamofire or Moya is overkill. It's true that `NSURLConnection`, the predecessor of `URLSession`, had an API that wasn't terribly elegant. The introduction of `URLSession` several years ago has changed this. The API isn't as sophisticated as Alamofire's or Moya's, but it does its job well. That should be your first choice.

Watch Out for the Defaults

There are several dependencies that have become almost default dependencies. These include libraries and frameworks for analytics, crash reporting, and logging. I consider it a dangerous practice to make a dependency a default dependency. I understand that your application needs crash reporting and analytics, but does that mean that you need to opt for Fabric or Firebase? Did you know that Apple also has crash reporting? You can even inspect the crash reports from within Xcode, pointing you to the line that caused the crash.

I'm not pleading that you should avoid Fabric and Firebase, but don't make them default options. Every dependency needs to earn its place onto the project. The same is true for the many SDKs that are available, such as the Twitter and Facebook SDKs.

Challenge Yourself

I’d like to challenge you. The next time you create a Cocoa application, a side project maybe, avoid relying on any third party dependencies. See how far you can go without including third party libraries or frameworks. Impossible? Think again. At its core, a developer is someone that solves problems. This is a problem you can solve. Believe me.

6 Automation

“A good developer is a lazy developer.” It’s a phrase I use several times in this book. I believe the term “lazy programmer” was first coined by Philipp Lenssen. The idea is that a programmer wants to avoid being repetitive whenever possible. But that’s not the only reason automation is important.

Automation can also minimize human error. Continuous integration, which we discussed earlier in this book, is an example of this. By automating a sequence of steps, we reduce or eliminate the number of manual steps a team needs to take to deploy a build to production.

Automation doesn’t need to be rocket science, though. The Cocoacasts website is backed up automatically. This is a trivial example of automation, but the result is that I have peace of mind and, when things hit the fan, I won’t lose any data.

Let’s start with the basics of automation, scripting.

Scripting

Many of us instinctively automate tasks we need to perform frequently. If you’re using an application launcher, such as LaunchBar or Alfred, or a snippet manager, such as TextExpander or Dash, then you’re already embracing automation.

With the release of Swift, however, you can take scripting to the next level with projects like Marathon, Swiftline, and Commander. Several frameworks, libraries, and tools have emerged that allow us to use Swift to create scripts and automate a slew of common tasks.

Before the introduction of Swift, I used Ruby to write small scripts that helped me automate a collection of mundane tasks. I’m slowly porting some of those scripts to Swift. You can use from a wide range of languages, including JavaScript, Python, and Perl.

Build Phases

Xcode build phases are ideal for automating tasks you tend to skip or forget. For personal projects, I use Realm's SwiftLint to make sure the code I write is consistent and sticks to a set of guidelines. SwiftLint is easy to set up and integrate with Xcode in a build phase. You can even trigger SwiftLine using fastlane.

Automated Testing

It's no secret that I'm a fan of automated testing. It isn't hard to set up and it builds confidence. Every commit you push to a shared repository triggers the test suite, immediately showing you if you've broken anything. This feedback loop allows you to confidently build a robust, stable product. Not having to think about running your tests is essential.

Documentation

If you or your team maintain a framework or library, then you need to make sure its documentation is up to date. As I mentioned elsewhere in the book, it's frustrating to work with outdated or incorrect documentation.

The good news is that this is another task that can be automated. If you properly document the code you write, tools like Realm's Jazzy can automatically generate documentation for you. Make it accessible to your team and you have access to up to date documentation with every commit you push to the library's or framework's repository.

Continuous Integration

Earlier in this book, I wrote about continuous integration and its benefits. It's a more sophisticated form of automation that can help you and your team stay on top of larger, complex projects with many moving parts.

Remember that one of the goals of automation is reducing the number of manual steps you need to take to deploy your product to production. From the moment a developer pushes the last commit to the shared repository, no human should tinker with the build that's deployed.

If you work at an agency or a development shop that manages dozens of products, then having a robust continuous integration solution isn't a

luxury. It's essential. No two projects are the same and every project has its own requirements, dependencies, and configuration. You don't want to remember those and you don't want to manually tweak build settings moments before you trigger a production build. Take your work and that of your clients serious and invest in a continuous integration solution.

fastlane

A few years ago, Felix Krause started developing a suite of tools for automating common tasks related to Cocoa development. This suite of tools is fastlane. At the time of writing, fastlane is a part of Fabric and Google acquired Fabric not too long ago.

The tools are open source and under active development. Felix is still very much involved in this amazing suite of command line tools and I recommend that you take a look at fastlane if it's new to you. My workflow doesn't rely on fastlane, but I do use it for generating screenshots, building, testing, and pushing builds to iTunes Connect.

Felix' suite of tools is an essential component in many companies because it also solves common problems developers face, such as creating and renewing provisioning profiles, creating code signing identities, and keeping them synchronized through Git. It's an amazing collection of tools.

Keep It Simple

Automation is great, but it needs to work reliably. For example, if your continuous integration setup isn't reliable, your team will start to lose confidence in it and more time is spent working on getting it to run than benefitting from it.

7 Privacy

Privacy and security are hot topics and rightfully so. What developers often don't realize is their role in protecting the user's privacy. In the chapter about security, I already wrote about privacy and what you can do to make sure the data of your users is kept from the prying hands of people and companies with questionable ethics.

Many parties are interested in the data of your users, for various reasons. From a developer's perspective, it's less important who's after this information. What's important is that the users of your application trust you with their information and they expect you to keep it private and secure.

Protecting the User's Privacy

Even though privacy isn't a fancy topic and it isn't top of mind for most developers, I hope you're at least keeping privacy in mind when you're building software. You sometimes have no other option, for example, if you're developing an application for a bank. But there is a range of less obvious application types that privacy is important for, for example, applications that have a link to the user's health.

Who Do You Work With

I already wrote about dependencies and how important it is to properly vet any dependency of a software project. This also impacts privacy. As I wrote earlier in this book, third party services, such as analytics and advertising, usually offer their services for free. Why is that?

You've probably heard the phrase "If you're not paying for the product, then you are the product." Some companies collect and sell information. They collect user information, such as browsing behavior and location data, and sell it to advertisers. While the services they offer are free, there's a catch.

It’s not always easy to discover which companies are legit and which ones to avoid, but if you, the developer, include a third party’s service in your application, then it’s your responsibility to make sure it doesn’t violate the user’s privacy. You’re on the hook if something happens with the user’s data.

You can remedy this by publishing a privacy policy for your application in which you explain what happens, or doesn’t happen, with the user’s information. This also means that you need to know how the third parties you work with use your application to collect information. This can be tedious. It’s ironic that it’s usually easier to include a third party SDK than to find out how the third party uses your application for their own gain.

As I mentioned earlier, I no longer include third party services, such as analytics and advertising in the projects I work on. That’s a conscious decision and I have the luxury to do this. Unfortunately, this isn’t possible for everyone. In fact, it may be rare.

Giving the User Control

One of the reasons for choosing for Apple’s ecosystem, both as a consumer and as a developer, are the values the company holds. Privacy is important to Apple and there are very few companies that follow its example in the technology industry.

App Review is a fine example. While App Review has received a fair amount of flak over the years and is often seen as an obstacle for developers, I applaud Apple for putting this barrier in place. After more than nine years, the company sticks with its ambition to keep the App Store safe and healthy.

Whenever I install an application from Apple’s App Store, I know that I won’t be installing a piece of malware that sends the contents of my address book to a remote server. App Review isn’t perfect or bulletproof, but it’s demonstrated that it does a pretty good job at protecting you and me from people and companies with dubious practices.

You need to give an application your explicit permission for it to access your address book, photos, and location. This is a good thing because it puts you, the user, in charge of managing your privacy. While this

solution isn’t perfect, it at least gives you the opportunity to be conscious of your privacy if that’s what you want. Some people don’t care about their privacy and that’s fine. The user is in charge.

Third Party SDKs

While you can’t always choose which companies you work with, you, the developer, are responsible for the user’s data, how it’s used, and who can access it. Over the years, many scandals have surfaced of companies that used devious tactics to get a hold of the user’s personal information, such as their contacts and location.

If you include a third party SDK in your application, then you should realize that you have no idea how the company providing the SDK is using your application’s data. This is something we often forget.

This is closely tied to the discussion about dependencies earlier in this book. Not only is a third party SDK a dependency that can cause havoc in your application, it also has unlimited access to your application’s data.

Choose Wisely

Consider the companies you implicitly, or explicitly, work with. I don’t want to imply that you shouldn’t trust Apple, Google, and Facebook, but you need to be mindful of the services your application and your business uses.

8 What to Do When You Inherit a Software Project

As an employee, freelancer, or consultant, you inevitably end up with a foreign codebase on your plate at some point in your career. In a way, inheriting a software project is much like receiving the keys to a house or car you don't know anything about. You hold your breath, afraid for what's about to come. But you're also a little excited, curious to find out what you'll be working on for the next weeks, months, or longer.

A new project can be overwhelming. Depending on its size and state, opening a project for the first time can be downright shocking. Don't let this intimidate you, though. Some things are hard, but that doesn't mean they're impossible.

First Things First

Take your time to explore the codebase and learn more about the project as you go. The first few days can be disorienting and it may feel as if you're not making any progress. And that's fine.

Talk to the Previous Owner

If the previous owner of the project is still around and able to do a formal handover, then you're in luck. This is a major advantage. Prepare a list of questions and fire away during the handover. Don't be shy. You're new to the project, which means there are no silly or dumb questions.

Is the previous owner not around, then try to get a hold of someone who worked on or is familiar with the project. That's the next best thing.

Source Control

Whenever a project is thrown in my lap, the first thing I do is clone the repository. Is the project not under source control or, even worse, someone tampered with the repository's history, then that's the first red

flag. It's possible the previous owners are trying to hide something. In that case, your first task is to create a repository for the project. If things go haywire, it's easy to retrace your steps.

If you're in luck and the project's under source control, then take a close look at the commit history. For now, the most important questions are "Who made the first commit and when was it made?" and "Who made the last commit and when was it made?"

The time of the first and last commits tell you a lot about the state of the project. You may even be able to contact some of the people that worked on the project. Source control is invaluable for any software project.

Compatibility

Depending on the software project, you need to make sure your development environment is compatible with the project's technology stack. A Cocoa application, for example, requires a machine running macOS with a copy of Xcode installed.

It may even be necessary that you need a specific version of macOS or Xcode to bring the project to life. As I mentioned earlier, if you can talk to someone that used to work on the project, then you're saving yourself a lot of time and frustration.

Build and Run

The moment of truth arrives when you compile the project for the very first time. Give it a try. Build and run the application on a device or in the simulator. This step is critical because it reveals some of the project's dependencies, such as third party frameworks and libraries, and it can also expose some early problems.

If the project uses a dependency manager, such as CocoaPods or Carthage, then you may need to install the dependencies first before you're able to compile the project.

Once you've successfully built and run the project, you've reached your first important milestone. You can now start to explore the project and collect information.

Collect Data

With the project up and running on your development machine, it's time to get to know your new best friend. It's important to understand how the project works and, more importantly at this stage, find out what its requirements are.

For a Cocoa application, for example, the deployment target is an important piece of information. It tells you what APIs you can use and which devices the application is compatible with. Are you dealing with a universal application or does it only support iPhone and iPod Touch? Has the project been updated for retina displays? Do you think that's a silly question? Think again.

Some projects have been collecting dust for years. This can sometimes mean that you need to make significant changes before you can submit a build to Apple's App Store. As of June 2015, for example, Apple requires new applications and updates to existing applications to include 64-bit support. If the project depends on a third party framework or library that hasn't been updated in years, this may be a problem.

Dependencies

Speaking of dependencies, you're probably starting to get a pretty good picture of the project's dependencies. It's time to take a closer look. Are the dependencies managed by a dependency manager or are they included in the main project?

With a bit of luck, the previous owner used a dependency manager to separate the dependencies from the main project. In that case, it's straightforward to verify the current version of each dependency and, at a later stage, update them.

In some cases, dependencies are included in submodules or, if you are less fortunate, they're mixed in with the project. The latter makes updating dependencies a pain. Migrating third party dependencies outside the main project with the help of a dependency manager is a worthwhile investment. It'll save you many hours and headaches.

Working on the Project

How you approach a project largely depends on the time you're allowed to spend working on it. If you only need to step in to fix a handful of bugs, then it isn't worth spending days or weeks familiarizing yourself with the codebase.

Are you tasked to implement a major feature, then it's key that you know how the application is architected and how it operates. If you're the new owner of the project and in charge of maintaining the project for the foreseeable future, then you have the luxury of investing time and energy to clean up the codebase, refactoring neglected parts, and making the project yours.

Document Everything

No matter what project you work on, it's important to document as much as possible. The README of the project is a good starting point. For larger, more complex projects, it can also be useful to document features or include an overview of the application's core architecture. It helps other developers become familiar with the project and it also benefits you, the maintainer, in the long run.

By commenting your code, documenting the project, writing sensible commit messages, and maintaining a list of bug reports, you have most of the tools you need to stay on top of your project.

And Beyond

What's next? That's a good question. In this chapter, we only scratched the surface. Starting work on a foreign or legacy software project can be messy. It may push you out of your comfort zone, but that's a good thing. Leaving your comfort zone from time to time keeps you sharp, forces you to be creative, and prevents you from becoming complacent.

9 Speed, Quality, and Technical Debt

The first version of a product can never be released soon enough. That makes sense. As long as designers and developers are working on a product that isn't making money, it's costing money.

But speed can come at a cost. To gain speed, you need to make sacrifices. And very often speed is traded for quality, resulting in technical debt.

Speed and Quality

In product development, speed and quality are almost always irreconcilable. Increasing the velocity of a project means compromising on quality. From a development perspective, this usually results in the creation or accumulation of technical debt.

Assume for a moment that you're the product owner of a software project. The first version of the product needs to include five major features. Each of these features takes a week to develop. The problem is that the client wants to ship the first version of the product in three weeks. What do you do?

You have several options. The most obvious one is dropping two features. Unfortunately, that's rarely something the client agrees to. Another option is cutting down development time of each feature by one or two days. This usually translates to removing anything that doesn't involve the implementation of the feature, such as code reviews, quality assurance, and testing.

Technical Debt

Technical debt can creep into a project without the product owner knowing about it. The development team usually knows, though. If the team is led by a senior developer and code reviews are baked into the company's culture, technical debt is easy to spot.

A fast approaching deadline can cause even the best to ignore technical debt. The symptoms start to appear when new features take longer to build than expected and regressions make their way into the product. These are the early symptoms of technical debt.

In advanced stages, technical debt takes a project hostage. Nobody wants to touch the project anymore. Features take ages to complete and are often compromised by technical limitations caused by technical debt. Days, weeks, or months of bug fixing are needed to stabilize the project. I'm not exaggerating.

In the meantime, the product itself evolves at a snail's pace. The speedy start that was once so important for the project's success has long been forgotten and, looking back, wasn't that important after all. It rarely is.

Focus

If the client wants to ship in three weeks, the correct answer is removing features. For projects with a long shelf life, you want to avoid technical debt at any cost. Technical debt is very much like a virus. It's hard to eradicate and, if it isn't treated in its early stages, it spreads out rapidly across the project's codebase.

Focus is essential to avoid technical debt. Instead of rushing out a feature, you take the time to craft something that stands out. This doesn't only relate to the final product; it also involves the development aspect of the feature.

You don't need to overengineer the solution, but you need to make sure the feature can grow beyond its current scope. You plan and anticipate how it can or could evolve.

How to Get Rid of Technical Debt

There's no miracle cure to rid a project of technical debt. You need to take action, though, if you want to avoid worse.

Refactoring

The least radical approach is refactoring the project, focusing on one problem at a time. If you're in luck, one round of refactoring is sufficient. However, most projects suffering from technical debt have many problems that need fixing.

As I mentioned earlier, technical debt spreads like a virus and that means many areas of the codebase are infected. I strongly advise against a single round of refactoring. Map the problems you plan to attack and spread the refactoring over several releases. This ensures the release cycle isn't blocked as long as the refactoring is ongoing.

In the meantime, make sure you don't introduce new problems. Hold off on new features if possible. Don't make the same mistake twice.

If you're working on a large project with years of history, be prepared to spend weeks or months refactoring. That's the price you pay for technical debt.

Clean Slate

The most radical approach is starting anew. This means you don't need to spend months refactoring. This isn't an option for every project. If you have the opportunity to start with a clean slate, though, you're in luck.

While it means that you need to implement every feature from scratch, it's an opportunity many developers would grab with both hands. Learn from your mistakes, or those of your colleagues, and create an amazing product.

I've worked on several projects that would have cost less if the project was rebooted. But that's often a very hard sell. The client doesn't see the problems the developer sees. They only start to see the symptoms of technical debt when deadlines are missed or features become very expensive.

Taking Shortcuts

Taking shortcuts rarely pays off in software development and most developers know this. But deadlines are often more important and

developers rarely have the authority to make decisions about the feature set of the product.

Part 5: Team

1 Code Reviews

Like unit testing, code reviews are often seen as a nice feature if there's some time left on a project. This misconception stems from inexperience. Once you start scheduling regular code reviews, you start to appreciate the value for anyone involved. If you work in a team, you'd be crazy not to take advantage of the knowledge, experience, and input from your colleagues.

Just Start

Code reviews don't necessarily need to have a goal. Pick a time and date to sit together with one of your colleagues, or as a team, and have him or her scrutinize your code. The simplest questions are often the most powerful ones. A common misconception is that code reviews are only useful to debug an issue or find problems in a project.

Code reviews are first and foremost a tool to improve yourself. I'll talk more about that later in this chapter.

Calendar and Agenda

It's important that a code review is focused and productive. That's why I always recommend to schedule code reviews in the calendars of everyone involved and to have an agenda before the code review starts. Every participant should be able to access and modify the agenda.

One of the participants needs to take the lead. That doesn't mean that only he or she can ask questions. It simply means that someone needs to make sure the code review respects the agenda and keeps the session productive.

Make Them Actionable

Code reviews can be very useful if they're scheduled at regular time intervals. If that's an option, I recommend compiling a list of actions at the

end of each code review and evaluating the actions at the start of the next session.

This also means that the participants of the code reviews need to keep each other accountable. If you commit yourself to refactoring a problematic view controller in a project, then you need to make sure it gets done by the next code review. If you ignore accountability, you risk turning the code reviews into a required meeting nobody enjoys or takes anything away from.

The goal of a code review isn't finding bugs. The benefits are much broader. You critically inspect someone's code and discuss it with an open mind. The goal is to learn and to improve the quality of the code. Finding and fixing bugs is a welcome side effect.

Keep Them Small

When a senior developer sits down with a junior developer, the mistake that's often made is biting off too much at a time. If the junior developer made dozens of commits since the last code review, adding hundreds or thousands of lines of code, then the code review is going to feel overwhelming. You need focus and direction.

Keep a code review small and focused, especially if you're training someone. The goal isn't to cover as much ground or code as possible. The goal is to go through a chunk of code, ask questions, and brainstorm ideas and alternative solutions. Even senior developers can learn from developers with much less experience.

Be Prepared

I already mentioned that an agenda is indispensable for productive code reviews. It takes more to be prepared for a code review, though. Are you unfamiliar with the project? Make sure you take some time to dive into the codebase to ensure you know what the project entails and how it's organized. Ask a colleague for help if you have little time.

If you have no idea how the code you're reviewing fits into the project, then you might as well review a random block of code you found on the web. This doesn't mean that you need to spend hours browsing the

codebase. Familiarize yourself with the project and zoom in on the code you're planning to review. That's a good start if you're short on time.

Tools

There are many, many tools available for organizing and structuring code reviews. I'm not going to mention any of these tools because I feel that they're secondary. Most of these tools have the option to *send* someone a code review. While I understand why this feature exists, the code reviews I'm talking about in this chapter are live code reviews, two or more people having a discussion about a particular piece of code.

You can still use a tool to keep code reviews organized and structured, and I recommend you do, but make sure you don't overlook the essence of code reviews. That's the main takeaway of this chapter.

That Hurts

Finding out during a code review that you introduced a bug can be painful and it can hurt your ego. To be honest, this is a facet of code reviews that I like. Why is that? Nobody likes to make mistakes and, from my experience, developers in particular hate to be pointed out that they made one. A fellow developer pointing out that you made a mistake can be hard to swallow.

It's therefore essential that code reviews are organized in such a way that the finger isn't pointed at someone. Everybody makes mistakes. No exceptions. If you don't believe me, then have a look at the crash reports of the projects you worked on in the past month or year.

It's a good sign if it stings a little when someone uncovers a bug you made. It shows that you care. Don't be arrogant and try to prove them wrong if you know they're right. You can only grow as a developer if you're willing to grow and learn from your peers. Learning from smart developers is one of the few shortcuts to speed up your career. Take advantage of this shortcut whenever you can.

Convincing Management

It’s often difficult to convince management of the value of code reviews. As with testing, it doesn’t seem to result in a direct return on investment. Even though the results aren’t directly measurable, the return on investment comes in many forms and shapes.

The most obvious benefit is an improvement of the code quality. A less obvious result is that developers feel more confident in the code they ship, and that isn’t only true for junior developers. Having a second pair of eyes look at a complex piece of code can do wonders.

If everyone enters a code review with mutual respect, it can also strengthen the team. This is a benefit that isn’t immediately visible and easily overlooked.

Frequency

How often should you schedule code reviews? That depends. If you schedule code reviews only once a month, then you may overwhelm developers with the amount of code that needs to be reviewed. Remember that small code reviews are most effective. To reduce the number of issues you find during a session, it’s a good idea to schedule a code review after the code you’re reviewing is covered by automated tests. This is a good strategy for two reasons.

First, you can inspect the code as well as the tests during the code review. Second, trivial issues that are caught by automated tests should already be resolved before the code review takes place.

There are several other factors you need to take into account, such as the size of your team and the type of software you’re developing.

But It’s Just Me

What do you do if you’re a team of one? That makes it more difficult, but there are several solutions. Even though you work alone as a freelancer or independent developer, or just as a hobbyist, you can always ask a fellow developer to take a look at your code. You can schedule code reviews and explore each other’s codebase from time to time.

This can be even more powerful than having code reviews in a team. Why is that? Each developer has its own style and this is usually even more the case for developers that work on their own. It can be eye-opening to see someone else's code or have someone critique your project.

One of the services I offer is code reviews and project audits. It can be useful to have a third party take a look at your code, ask questions, suggest alternatives, and, from time to time, find bugs.

Give It a Try

Code reviews have many benefits. If you're new to code reviews, then I encourage you to give it a shot. I recommend running a trial for at least a few weeks or months. Give your team the chance to become familiar with the process and tweak it as you go. You won't get it right from the start.

2 Adopt a Style Guide

Even if you're working on your own, in a team of one, adopting a style guide can be very helpful and useful. It ensures that you stick to a set of guidelines you define. The result is that your code is consistent and easy to read.

Why Have One

The most obvious benefit of having a style guide is promoting consistency. It ensures that the code you and your team produces is consistent. This benefit alone is enough to consider adopting a style guide. But it's only the tip of the iceberg.

Another immediate benefit of having a style guide is improving readability. When I browse a foreign codebase, it usually takes some time to become familiar with the author's style. This includes naming conventions, comments, spacing, and even the use of curly braces. A style guide normalizes this, making it easier for developers of the same team to read each other's code.

A third subtle benefit is reducing friction within the team. There's no internal discussion between team members about the use of curly braces and it ensures developers don't need to think about their style. They simply stick to the conventions defined in the style guide.

Automation

I've worked in teams with a formal style guide that wasn't enforced in any way. Team members were simply expected to stick to the style guide. If you decide that a style guide can benefit your team, then I recommend you also invest in automation to enforce the rules defined in the style guide.

This is very easy to set up if you're using Xcode. Adding a build phase to the project that triggers a shell script or a command line tool should do

the trick. Be careful that the style guide you enforce locally is up to date. If you open an old project, make sure it uses the most recent version of the style guide.

Rough Transition

Making the transition from no style guide to a clearly defined specification can be rough for some team members. I advice enforcing the style guide for new projects only unless your company runs a product business.

You need to avoid that developers are spending hours or days updating projects because of style violations. Not only is this time-consuming, it almost always introduces bugs if there’s a large number of changes that need to be made.

Adopting a style guide is a big change for many developers. Try to ease the pain by gradually enforcing the style guide.

3 Working In a Team

> The whole is greater than the sum of its parts. â€” Aristotle

Working in a team can be a joy with many benefits, but it can also be a challenge. Every team member has a responsibility towards the team, and this determines in a significant way how effective the team is.

Hire the Right People

Growing a company is challenging for a wide range of reasons, hiring is one of them. Young companies are often still discovering themselves, and it may not always be clear who you want to hire. You know what their resume should look like, but how do you know whether they fit the company?

If you aim to build a reliable, efficient team, you need to hire people that excel in a team. You need people that are ambitious *and* eager to work in a team. Those qualities don't always go together, though.

What's most important is that the person you hire fits the company culture. Finding someone that ticks every box is hard, and it takes time, especially in the technology industry.

Leadership

A team needs a leader, even small ones. One person needs to make decisions and have the last word when push comes to shove. A leader often has a significant impact on the team and the team's ethos.

Having a healthy mix of junior and more senior developers is a good strategy. Attracting someone who has experience working in and building a team is a big plus.

Communication

Elsewhere in the book, I write about focus and productivity. At times, it's necessary to put on your headphones and focus. That's unavoidable if you want to get work done.

What I've also noticed is that this culture of headphones can sometimes result in a lack of communication. Talking to your teammates is as important as getting work done, especially if you're working towards the same goal.

You can solve this seeming conflict by organizing team meetings that focus on team building. Take the afternoon off for a team event that doesn't have anything to do with work. Or organize a weekly lunch to talk about a technical topic. Start a discussion and leave room for everyone to share their opinions.

Give It Time

It takes time to turn a group of people into a team. It isn't necessarily a natural process and don't expect it to be. A solid team isn't built overnight nor does it happen by accident.

A good team strengthens when it gets tough. When something goes wrong, there's usually one person to blame, but you take the hit as a team. It's a team effort, and you need to avoid that someone is singled out, taking the blame. You're in this together. People should be allowed to make mistakes.

Respect

Not everyone is cut out to work in a team and you can't always choose the other members of your team, which can cause friction. That's why hiring is such an important aspect of a growing company and something many struggle with. I'm sure you can think of a few examples.

Respect is a key ingredient of a healthy team or company. This may seem obvious, but it's not always easy. You can't choose the people you work with. The larger the team, the more likely it is that you're going to meet someone you don't see eye to eye with. This doesn't need to be an issue, though, as long as you respect one another.

When Things Hit the Fan

Things will go wrong if you're solving complex or ambitious problems. Accept that this is inevitable. But, as I mentioned earlier, the team takes the hit.

You obviously need to investigate the issue and find out who made a mistake, but remember that it's a team effort. You're in this together.

Some team leaders or managers try to cover problems up. I recommend taking the opposite approach. Analyze the problem and investigate, as a team, what went wrong. It's in the tough times that teams are built or fall apart. That's when you discover how much you can rely on your team and your fellow team members.

Ownership and Responsibility

Taking ownership and responsibility are aspects of working in a team that aren't talked about very often. However, the most performant teams and companies I worked with are the ones that embrace these values.

There are dozens and dozens of small tasks in a company. Having a committed team makes handling these tasks much easier. Knowing that someone takes care of a particular task you don't need to worry about is one of the most enjoyable aspects of working in a team.

That's also one of the aspects I enjoyed most when I was working as a subcontractor. While I've worked with many project managers over the years, I had my favorites. An experienced project manager is core to an effective and efficient team. They're the glue that binds clients, designers, developers, and testers together. An experienced project manager owns the project and takes the final responsibility for every aspect of the project.

And developers need to do the same. If something goes amiss, then the person responsible for that aspect needs to step up and take care of the problem.

When something goes haywire and people start pointing fingers at each other, then there's a problem that needs to be addressed. It should be

obvious who's responsible and who needs to take action. That person shouldn't be blamed for what went wrong, though. It merely eliminates the discussion who needs to step in and fix the problem.

Share and Ask

I'm sure it's obvious that you should share what you know with the team and learn from your teammates. It's one of the shortcuts in software development that you should take advantage of. Stack Overflow is great, but nothing replaces a healthy discussion with your teammates about a technical topic or a problem you're trying to solve.

And don't forget to include designers and managers. I've had many interesting discussions with team members that weren't technical. There's more to software development than writing code.

4 Being a Leader

As a freelancer, I don’t have much experience leading teams, but I’ve been part of several teams as a subcontractor. Leading a team and acting as a leader can be challenging for developers because they’re not always asking for this type of promotion. As a company grows, teams are formed, and a team needs a leader.

Listen and Be Open

A good leader listens to the people he or she works with. It can be frustrating to be part of a team in which your voice isn’t heard. It doesn’t mean that you need to take action based on what you hear, but you need to make sure every member of the team knows that he or she can approach you, talk to you, and know that you understand them.

It’s equally important to be open in your communication. Always make sure that there are no unresolved issues in your team that create tension. Talk to your team members, as a team, and in one-to-one meetings. Everyone needs to be up to date about what’s going on in the team and the company. If a team member or employee is unaware of an important event or announcement, then that person may feel left out or not part of the team.

Modesty, Humility, and Respect

The most memorable leaders I worked with are modest and humble. They can bring the best out of you, taking advantage of the strengths of each member of the team.

Developers that don’t feel comfortable leading a team sometimes have the tendency to please the team they lead in an attempt to earn their respect. If you climb the ranks in a company and are promoted from team member to team leader, then earning that position can be challenging. But you don’t earn the respect of your team by attempting to please them.

Work as a Team

Most of the team or project leads I've worked with had too much on their plate. That's a very, very common problem for people leading a team or a company. Their workload isn't always the cause of the problem, though. Two causes are very common.

The first cause is wanting to do everything yourself. Inexperienced leaders, especially those with a healthy dose of pride, find it hard to ask the members of their team for help. You won't be able to do everything yourself and you need to learn to delegate tasks to the members of your team.

This can have a dramatic impact on your workload. At first, it can feel as if you're asking someone else to do your job, but that feeling quickly disappears once you discover that you have more time for other tasks, such as organizing your team and setting up one-to-one meetings with the people of your team.

Delegating is something you need to learn and it also means you need to know the people on your team very well. Each member has their strengths and weaknesses. It's up to you to leverage this diversity and use the strengths of your team to your advantage.

The second cause is closely related to the first. Handing control over to other people can be scary, very scary. You're the leader of your team and that means that you carry the final responsibility. As a result, inexperienced leaders have the urge to make sure everything is exactly as they want it to be, as if they did it themselves.

This approach isn't healthy. First, the result is that you increase your workload for no good reason. Second, you may give other team members a feeling of inadequacy, not being good enough. It can sometimes be good to let things hit the fan. Everyone learns from such an experience.

If you still struggle with this, then you may want to embrace code reviews or roundtables. These can be very effective strategies to build your team, show them how things should be done, and make sure everyone is on the same page.

Working With People

As a developer, you're used to finding and implementing solutions to problems. It's not that straightforward if you're working with people. Developers share a passion for solving problems, but they're human beings and each of us is different. This means that you need to approach every member of your team or company a little differently.

This can be one of the most challenging aspects of being a leader because it means you need to rely on your intuition. You need to be able to read the people on your team and rely on your people skills.

Some of your teammates are extroverts and you always know what's on their mind. Others are quiet, shy, or not eager to share their opinions with others. It takes leadership to ensure the voice of the quiet is also heard by the extrovert.

Follow Your Gut

Even if you sign up for a crash course in leadership, it takes time for most people to become a leader and be seen as one. You won't always know what to do in certain situations. How do you handle a conflict within your team? There's no one or right answer.

As the leader of your team or company, you're in the best position to assess the situation and make a decision that feels right to you. It's possible that you take the wrong decision, but that's how it is. You win some and you lose some. Learn from your mistakes, be transparent about them, and move on.

Lead

Listening to the people you work with and asking for their input is essential to building a healthy team or company. But always remember that you're the leader and you have final say when decisions need to be made. And that's what a team member expects from its leader. When push comes to shove, you need to show that you're the leader and the man or woman taking the decisions.

This won’t always be easy, but it’s necessary. It’s as simple as that. You won’t always know whether that decision is the right one, but that’s how it is. That’s what sets a leader apart from a team member or an employee.

Part 6: Career

1 Open Source

The concept of open source has been around for decades, but platforms like GitHub have accelerated the growth of open source initiatives. Most of us can't imagine a world without open source software.

Open source software can be a powerful instrument to promote yourself as a developer. Many respected developers in the Cocoa and Swift communities have gained name and fame through their open source projects, think Eloy Dur□□n, Mattt Thompson, and Ole Begemann.

Start Small

If you're new to open source, then I don't recommend diving in head first. Start small. I'm sure there are several open source projects you make use of in your own projects. A good place to start is to explore the documentation of these projects. That's right. The documentation.

You'd be surprised by the number of open source projects that have outdated documentation. This isn't surprising since it requires a lot of work to maintain an open source project. Because you're already familiar with the project, you can browse the documentation and fix any issues you find on your path, including typos and grammatical issues. While this may seem nitpicky, the author of the project will thank you for it.

Clone the project, make the changes, and submit a pull request. Make sure you read the guidelines for contributing to the project first. Many projects have a code of conduct to make contributions easier and reduce friction.

Taking the Plunge

You may already have a library or project you want to share with the community. Sharing something with the world can be scary, though. Make sure you've cleaned up the project and don't share any personal or

confidential information. When you share a repository with the world, you give everyone access to its history.

For example, if you committed a private key or token at some point and removed it later, then anyone with access to the repository can retrace your steps and access that information. Remove the project history if you want to play it safe. Once the information is public, there's no way back.

Documentation

Most of us don't spend much time documenting personal or internal projects. That's a decision everyone needs to make for themselves. An open source project without documentation, however, won't get much traction. Not only is it difficult for anyone interested in your project to get up to speed, it's not a healthy sign.

Every time I look for a third party library to include in a project, I immediately discard repositories without a proper README. It only takes a few minutes to create a README and an additional hour to create an installation and getting started guide. Not only does your project look more appealing, you help anyone interested in your project.

You can take it one step further and add a sample application that shows developers how to use your library. Anyone new to your library can play with the bundled application and see for themselves how to use your library and integrate it into a project.

Taking It Seriously

The commitment of some developers is admirable. They cover their project with unit tests, set up continuous integration, and write pages and pages of documentation. This is usually a community effort and that's what's wonderful about open source software. Once your project gains traction, developers automatically start giving back with small, or large, contributions.

These contributions are very welcome, especially if you have limited time to spend on your open source project. If you're a freelancer or an independent developer, then it can be a breath of fresh air to collaborate with other developers on a project. That's a bonus you get for free.

Giving Up Control

For some developers, the hardest part of open sourcing a project is giving up some of the control they have over the project. You can decide to manage everything yourself, but that isn't feasible if your project becomes popular. It also goes against the spirit of open source software.

To avoid that your project becomes messy and an odd collection of coding styles, you should adopt and enforce a style guide. This isn't difficult to implement. When I submitted my first pull request to the CocoaPods project, I immediately received automated feedback about the coding style I was using. I took a look at the project's style guide, made a few changes, and submitted an update. Having a style guide that's automatically enforced helps to keep your project consistent and it keeps the maintainer(s) sane.

Taking Responsibility

One of the most challenging aspects of an open source project is keeping it up to date. It takes time and effort to keep your projects up to date. Automation can make this much easier. Know that several hosted continuous integration solutions offer a free tier for open source projects. If you're serious about your project, then that may be worth looking at.

And remember that you don't need to do everything yourself. For Cocoacasts, I create many, many projects for the tutorials I write and record. It's challenging to keep them up to date. If a reader notifies me that a project hasn't been updated for the latest version of Swift or Xcode, then I suggest that they submit a pull request. That's not unreasonable. Is it?

Open source is a community effort and everyone involved in a project should take responsibility.

2 You Are the Constant in Your Career

One of the reasons for writing this book is that it allows me to talk about subjects that are related to programming, software, and the life of a developer. Even though they're related, they're often overlooked or ignored. This is especially true for young developers. Don't feel bad, though. It's natural and maybe even a necessary part of your evolution as a developer.

Putting Yourself First

Most of your time is spent working for someone else, a boss, a manager, or a client. The technology industry is known for long days and weeks and sacrificing time with friends and family. But don't let that distract you from what's important. What's important isn't the project you're working on or the client you're building a mobile solution for. What's important is you. Why is that?

Many developers change jobs every few years. There are many reasons for doing so. That's not the point, though. No matter where you end up next, there's only one thing you take with you. You. You're the constant in your career. This may sound like a simple platitude, but it isn't.

There have been times that I worked for the same client for several years. It was good money, but I didn't feel I was reaching the goals I set for myself. I felt that I was stagnating as a developer, and that's not a good thing for a freelancer. The space is evolving at breakneck speed and you can't afford to stagnate.

Setting Goals

Every now and then, I set goals for myself to keep me motivated, accountable, and to make sure I can objectively validate that I'm evolving as a developer. I'm sure you have a few things on your list that you want to learn, some day. Who's keeping you accountable? How long have these items been on your list?

Learning Requires an Investment

If you're in luck, then you have the time to learn new things. You may even have a boss or manager that explicitly asks you to investigate a new technology to pick up.

Learning something new requires you to invest time and energy. You need to allocate time to seriously look into and experiment with the technology you're learning.

Take Care of Yourself

Earlier I mentioned that long days and weeks are not uncommon in the technology space. That's fine as long as you don't push yourself too hard. Everyone has limits. I know what it's like to feel invincible when you're in your twenties. That feeling of invincibility disappears gradually as you approach your thirties and forties. Burning the midnight oil isn't something I've done in a long time. I'm too old for that.

I've been around developers for long enough to know and understand how tempting it can be to only focus on the job. We live in times where work is a valid excuse for ignoring other important aspects of our lives. Working out or going for a simple walk is often more important and sacrificed as a result.

It should come as no surprise that many of us suffer from a slew of illnesses and subtle health problems, from obesity, diabetes, and back problems to chronic fatigue and bad sleep. The underlying problem is surprisingly often related to work, directly or indirectly.

If you're in your twenties, then you probably think that I'm talking rubbish. If you're in your thirties, then you might already think a bit differently. If you're in your forties, then you're probably nodding your head.

Looking Back

I'd like to end this chapter with a simple assignment. In December, I always take the time to look back at the past year. The last weeks of December are ideal for this exercise. Work is slowing down and many

people are taking some time off to spend with friends and family. But you can do this exercise whenever you like.

Take a piece of paper and write down what you've accomplished this year. Don't use a computer or another electronic device. Use a pen or pencil and a piece of paper. Which goals have you accomplished that moved the needle for you? Not for your boss. Not for your clients. For you.

What have you learned? Who have you met or spoken with that impacted your way of thinking? What books have made an impact on the way you think about your work as a developer? Give yourself ten to fifteen minutes. Don't stop too soon. The best stuff comes out after a little while.

Are you happy with the list you've compiled?

3 Build That Application

Do you have a folder on your machine that's filled with unfinished projects? Be honest. I know you do. I have dozens of unfinished projects. Each of these projects started its life as an exciting idea, an idea I absolutely needed to execute on.

When a fresh, exciting idea enters the mind of a software developer, it's hard to resist the madness. These projects often start with a few days or weeks of hard work, very little planning, no business plan, and only a vague idea of what the final result will look like.

These unplanned projects are great. I love working on them. The sad part is that they almost always end up collecting digital dust, joining dozens of other unfinished projects. Building is easy. Shipping is hard.

Build and Ship

> Real artists ship. â€" Steve Jobs

Steve Jobs famously said that real artists ship their creations. You'll find this quote more than once in this book. It's easy to come up with a compelling idea, spend a few days or weeks building the product, and shelve it, elegantly wrapping it in a promise to finish it when you have more time.

It's easy, but, more importantly, it's safe. It's safer because few people are willing to show the world what they created. They fear the opinions of others. They fear people are going to judge their product and, even worse, its creator.

Show What You Can

When you're applying for a job, showing that you have relevant experience is often essential. Not too long ago, several years of

experience were mandatory for a job as a software developer. Nowadays, employers are looking more and more at the portfolio of potential candidates. This is especially true in software development.

People new to software development or graduates looking for their first job may find it challenging to prove that they have experience developing software. The truth is that they often donâ€™t have the necessary experience.

How do you solve this problem? How do you break this vicious circle? Don't wait for that first job or project to start gaining experience. Show potential clients or your future employer that you *have* experience building software. It's never been easier to get started with software development. What's stopping you from creating?

Stay Ahead

While it's fine to learn as you go, it'll certainly help you if you know the basics of the most common tasks of software development. Building *and* launching a software product is quite an involved process. It doesn't stop the moment you have successfully built and deployed your first iOS application on your iPhone or iPad.

Have you thought about beta testing, launch images, application icons, and localization? If you can show your future employer that you have thought of, built, and shipped a software project that's polished and carefully maintained, you're almost certainly what they're looking for. An interview is almost guaranteed.

Rinse and Repeat

Apple's App Store is filled with abandoned applications. Frequent application updates are a sign to your customers that you're building something you care about and customers will automatically feel that you also care about them.

It isn't necessary to include a major feature in every update you ship. Improving your application's stability by fixing bugs is just as important as focusing on features.

Localization is another aspect that’s easily overlooked. Did you know that Japan is one of the most profitable markets for mobile applications? Consider working with a translator to translate your application to Japanese. It won’t cost you much and you’ll learn a lot from the experience. Your Japanese customers will thank you for it.

Build That Application

I encourage you to start building and, more importantly, to start shipping. Your first application doesn’t need to be perfect. What’s important is to get it in the hands of your customers. Create something useful for others. Refine what you’ve created and listen to your customers. What’s stopping you? What are you waiting for? Gain experience by building and shipping.

4 Protect Your Productivity

How productive you are as a developer isn't only the result of your habits, your talents, and your experience. Your environment plays a key role, especially if you work in a team, a company, or a shared workspace.

Some developers thrive in a noisy or busy environment, but those are usually exceptions to the rule. That rule is simple "Don't interrupt the developer."

Working From Home

Working from home is a luxury if you have the discipline to set clear boundaries for yourself. Unfortunately, it's a very, very common trap many people fall into, not only developers. When you make the switch from working as an employee in a large office to working from home in a small home office, if you're lucky to have one, you may naively think your productivity is about to skyrocket. The opposite is almost always true, especially in the first few days or weeks.

Making a drastic change, such as moving into a home office, means that you need to create new habits. Most of us believe it has everything to do with discipline, but that's only part of the equation. Take the time to create a schedule that works for you and the people you work and live with. That may mean that you start early or put in a few extra hours in the evening when the kids are asleep.

You need to define boundaries, not only for yourself, also for the people you live with. Working from home is still work.

Working In an Office

Working in an office has its pros and cons. The advantage is that everyone knows and understands that you're at work to, well, work. Make it clear that you don't want to be disturbed. Protect your productivity. This

may sound harsh, but it’s essential if you want to get meaningful work done.

As a developer, your most important asset is your attention. If you’re working on a complex task, it takes time to gain momentum and be productive. Every time you’re interrupted, you need to start from square one. Not only is this frustrating, it can be mentally taxing.

I once worked in an office that had a simple rule that said that a developer with its headphones on shouldn’t be interrupted. A room filled with people wearing headphones isn’t uncommon in the technology industry where open offices are the norm. You’d be surprised by the number of developers that wear headphones without actually listening to music. It’s a trick I’ve used many, many times. This is especially effective if your headphones have noise cancellation.

While this rule looks great on paper, it doesn’t hold up in most situations. Everyone is interrupted dozens of times a day. The number of productive hours we put in is much lower than you’d think. That’s the reality we live in.

Minimizing Interruptions

Slack and email have become indispensable in most companies, large and small. They’re great tools for communication if they’re used wisely. They’re not most of the time. They’re surprisingly often used as distractions instead of productivity tools.

Make sure you’re not distracted by a slew of, often unimportant, notifications. If you’re constantly interrupted by notifications, then it’s clear you have no intention of getting work done. Don’t look for productivity hacks if you’re not serious about the most fundamental elements.

Find a Quiet Place

If you have a deadline to meet or you absolutely need focus, then looking for a quiet spot in the office may be your last resort. This isn’t a solution long term, but it’s sometimes your only option. If you’re in luck, you may

find an empty meeting room, but a couch or the kitchen is fine too. Don’t forget to bring your headphones.

Deal With It

Distractions and interruptions are part of the world we live in. Working in the technology industry means that you’re in the eye of the storm. There’s no perfect solution unless you work from a home office where you dictate the rules.

5 Building Your Portfolio

A portfolio is one of the most important assets of any developer with ambition. Mobile applications, for example, are so commonplace that potential employers or clients will want to see what projects you've worked on in the past. Aspiring developers often ask what they should show if they haven't created any applications yet. The answer is simple. Start creating.

If you're learning the ropes of software development and you don't have any clients yet, then what are you doing all day? You don't need to learn everything there is to know about software development to start your first project.

Scratch an Itch

While not every idea will be a runaway success, the itches you have are perfect for your first projects. If you're just getting started, then it's important to choose a relatively easy problem to solve.

The first application I shipped was an iPad application that allowed users to import and tag images from an Aperture or iPhoto library. While I did eventually publish the application, it was a bit too complex for a first project.

Start small and build larger, more ambitious projects as you grow as a developer. By biting off more than you can chew, you may lose motivation or become discouraged.

Setting Goals

The primary goal of your first project should be simple, creating and publishing an application that provides value to people. You can set secondary goals, such as trying to monetize your application or climbing the charts of the App Store. Those goals are less important, though.

Monetizing your application is only possible if you share it with the world, which means you first need to build and ship an application.

I love running. Every time I go for a run, I have three goals. The first and most important goal is to enjoy the run. The second goal is to finish the run. And the third goal is to beat my best time.

In the early stages of your career, it's important to approach software development with the same mindset. Make sure you enjoy what you're doing. If you don't, you won't stay committed to learning your craft. It shouldn't feel as a struggle. I dare say that it shouldn't even feel like work.

> It shouldn't feel as a struggle.

The second goal should be to complete the project you're working on. Once you commit to a project or idea, it's important that you stick with it. Steve Jobs famously said "Real artists ship." As a developer, you're creating. That also means that you need to ship your creations.

> Real artists ship. â€" Steve Jobs

The third goal is the proverbial cherry on the cake. That goal may be hitting the top charts in your application's category or reaching a certain number of downloads. That's up to you.

Why am I giving you this advice? And what gives me the authority to give you advice in the first place? Ask any developer how many projects they started but never finished. Starting projects is fine. But sometimes you need to ship. Do you want to be taken seriously as a developer, then you need to be able to show that you can start and finish something.

> Do you want to be taken seriously as a developer, then you need to be able to show that you can start and finish something.

Whenever I interview a developer for a project, I ask them about their previous projects. Your future employer or clients will do exactly that.

Even if you plan to start as a junior developer or an intern at a company or agency, it plays to your advantage to show one or more projects you started and finished successfully. It makes a world of difference. Why? Because finishing something is harder than it seems.

If software development is part of your DNA, then you'll be able to start and complete your first software project. Most developers have a hard time stopping themselves from creating cool software or using the latest, fancy libraries. Despite this drive to create, it's equally important to finish and ship. I cannot emphasize this enough.

Learn the Basics

A question that comes up often is whether you should dive in head first or learn the basics. The answer is, "Both." It's key that you know the foundation of the language you're using, but it's also important that you start creating as soon as possible. I explain this in more detail in the first chapters of this book.

Apple's platforms are so accessible that I guarantee you that you can create a basic Cocoa application the day you start learning Cocoa development. There's no need to read The Swift Programming Language from cover to cover. It'll discourage you since you started with Cocoa development to create cool things. My suggestion is to choose a guide that teaches you Swift as you learn the ropes of software development. The sooner you see the bigger picture, the sooner you'll feel comfortable on the path you're taking.

There is one big *but*, though. If you only focus on creating things and don't take the time to learn the more intricate aspects of the platform, you'll end up creating bugs, missing best practices, and, in the end, being frustrated why something isn't working. Remember what I wrote earlier, "Don't ignore your foundation."

Learning a human language is easy at first if you only focus on memorizing words and sentences. Your memory is only able to take in so much, though, and what are you going to do if you run into a situation you don't know anything about?

Most language courses start with simple sentences to give students the feeling that they're making progress. Sooner rather than later, the teacher teaches them about spelling and grammar, explaining the sentences they learned from a grammatical point of view.

The gist is that you need to become familiar with the basics, the foundation, if you want to grow as a developer. It isn't always as interesting as learning the cool things, but it's just as important.

But Start Creating

No matter what path you choose or what language you learn, it's important to start creating. Set a goal, pick a problem you want to solve, and build a solution. That's what software development is about.

6 How Badly Do You Want It

It isn't easy for inexperienced developers to find an attractive job without years of experience or an impressive portfolio. That shouldn't be an excuse, though. It's never been easier for developers to build a portfolio, gaining experience along the way. In this chapter, I show a few examples of how you can build a portfolio and gain experience to help you find clients or impress your future employer.

Contribute

Whenever I vet a developer, I take a look at their past work. Open source contributions are ideal for this. It isn't necessary to build an open source project from the ground up, though. Engagement and taking responsibility in existing open source projects is a good first step.

Many well known developers have gained name and fame by creating and maintaining open source projects. Eloy Durán, for example, had years of experience as a Ruby developer and missed a reliable dependency manager for Cocoa. Instead of complaining about it, he created CocoaPods and turned it into the most popular dependency manager for Cocoa development.

Open source projects often die a quiet death due to the lack of contributors. But another common reason is the absence of good documentation. Nothing stops you from creating a pull request to update the README of an open source project to polish its documentation. Developers new to the project will thank you for it and so will the maintainers of the project.

Build

It's never been easier to create software. Getting started with software development isn't that hard. What's stopping you from creating your first mobile application? If you're looking for a job, then your future employer *will* ask you about your portfolio.

Finding clients is hard without a portfolio. Would you hire an architect who has never built a house before? The fact that you're looking for your first job as a software developer or freelancer isn't an excuse for not having a portfolio with one or two applications. Build that application.

Maintain

Being a software developer also means that you're responsible for maintaining what you create. It isn't necessary to ship a new feature every month. What's more important is making sure existing users are happy and stay happy.

Take a look at the crash reports of your application. Are there any issues you can fix? What are people complaining about? Could you improve the onboarding experience?

Being able to show that you have the skills to build *and* maintain a software project goes a long way. Not only will your future employer look for these skills, you also need them to excel as a software developer.

Plan and Manage

If you want to take it up a notch, then scheduling releases and adding features is the next step. The idea is to start simple, build a modest user base, and improve the application over time. Listen to the users of your application, sift through the feature requests, and decide which features make the cut. Remember that you manage the project, not the users of your application.

Test a new feature before releasing it to the public. Involve your existing users by distributing a beta build of the application through Fabric, Hockey, or Apple's TestFlight.

Adding support for multiple languages is another great improvement that underlines your commitment to the project. This may involve hiring a translator.

Gain Experience

Getting started with software development is easy. Starting from zero and publishing an application on Apple's App Store requires effort and perseverance, though. It means that you cannot quit halfway when you don't feel like finishing the project. Every developer has a slew of projects they started working on and never finished.

Finishing is hard, but that's exactly what your future employer or potential clients want to see. There's nothing more enjoyable than starting a new project, but it takes grit and tenacity to finish a project and click the submit button to publish your creation on the App Store.

It's to finish a project and ship something people can use. It can be scary to share your work with your peers, but that's what it takes to succeed.

Be Yourself

There's no need to compete with your peers. If you continue to do what you love and what you're good at, you will get where you want to be. But remember that you only get what you work for. If you aim for mediocracy, then that's what you'll get. If you aim for the best …

How Badly Do You Want It

Whether you succeed depends on the question "How badly do you want it?" Are you willing to work nights and weekends? To make a dent, you will need to go beyond showing up.

> Showing up is overrated. Necessary but not nearly sufficient. — Seth Godin

It's important to be honest with yourself and define what you expect from your career as a software developer. Gary Vaynerchuck is spot on when he says "If you live for weekends or vacations, your shit is broken." Is that you or are you aiming higher?

> If you live for weekends or vacations, your shit is broken. — Gary Vaynerchuk

What About You

Are you motivated to get off your butt and make awesome stuff? What are your goals and dreams? You don’t know? Then that’s the first thing you need to sort out.

7 Freelancing and Subcontracting

Making a living as an independent freelance developer is great, but it can be a tough job at times. What I enjoy most about freelancing are the skills you learn to master, or better, are forced to master. A team of one has its upsides and its downsides.

Freelancing and Subcontracting

As an independent freelance developer, you spend a significant chunk of your time doing tasks that don’t involve programming. If programming is what you want to focus on and you have the ambition to run your own business, then you may want to consider subcontracting?

What Is Subcontracting

> A subcontractor is an individual or in many cases a business that signs a contract to perform part or all of the obligations of another’s contract. â€” Wikipedia

The above definition sums it up nicely. As a subcontractor, you are hired by another company to carry out work for one of their clients. Why would a company do that? There are a number of reasons. The most common ones are a temporary shortage of manpower and a lack of expertise in a particular field.

Why Consider Subcontracting

It’s true that, at times, it feels as if you’re an employee of the company you work for. Subcontracting isn’t for you if that’s a compromise you’re not willing to take. I can see why you don’t want to go that route as an independent freelancer. As a subcontractor, you run a business while, at the same time, you’re working for another company. There are several notable benefits, though.

Change of Environments

One of the advantages of subcontracting is the change of environments. I don’t mean a change of scenery, but a change of development environments. By working at different companies, you come in contact with a wide range of people and company cultures. You learn the pros and cons of the tools they use, the workflows they apply, and the best practices they live by.

Even if you’ve been programming for years and years, you learn something from every company you work for. There’s always someone who’s smarter than you and who you can learn from. This is a benefit of subcontracting that’s often overlooked. As a developer, it’s one of the most important benefits if you ask me.

If you enter a contract with the mindset of knowing everything you need to know, then this free benefit isn’t for you. That said, if you ever come at a point where you think you know everything there is to know about your craft, then it may be time to evaluate your current situation. I have yet to meet someone who knows everything about their craft. The more you learn, the more you realize how little you know.

Focus on What You Love

As I mentioned earlier, an independent freelance developer needs to master a wide range of skills to run a successful business. Finding leads, talking to customers, and writing proposals and specifications are but a small subset of the things you do as a freelance developer and business owner. Some people love that mix of responsibilities. If that’s you, then subcontracting may take some getting used to.

But if you started freelancing because you enjoy programming more than anything else, then you may not like those responsibilities all that much. In that case, subcontracting could be a great fit for you.

In larger companies, developers can focus on what they do best, programming. A project manager takes care of client relationships, the sales team makes sure new projects are lined up for you, and the operations team handles the infrastructure.

Choose Your Clients

The company you work for as a subcontractor is your client. The company's clients are not your clients. Before you decide to subcontract for a company, take a look at the company's portfolio. It's fine to be picky, especially if you have a portfolio you can be proud of.

Subcontracting or Employment

The line between subcontracting and employment may seem thin at times. This is especially true if you've been working for the same company for months or years. There's nothing wrong with that, but it's important that you feel comfortable in that situation.

If subcontracting begins to feel like employment without the benefits of being an employee, then it may be time to find another project at a different company or choose a different path altogether.

If this really bothers you, then being an independent freelancer may be a better fit. Nothing stops you from mixing subcontracting with working as an independent freelancer. Many freelancers do. The line between subcontracting and freelancing is sometimes pretty thin.

Both paths will teach you more than you can imagine. But make sure you don't become complacent. That's my one advice. If you choose a particular path because it makes your bank account happy, then it's time to reconsider your choices and priorities.

Being Picky

Elsewhere in this book, I explain which steps you need to take when you inherit a software project. If you're working as an employee at an agency and your company acquires a client from a competing agency, this often means that you also acquire one or more existing projects. This is usually part of the deal. As an employee, you have very little say in the matter. Inheriting a software project is rarely a gift.

Things are a bit different as a freelancer. You have the luxury and freedom to choose who you work with. For example, I no longer take on existing projects because it usually means several days, weeks, or months fixing bugs, making minor improvements, and spending a lot of

time fixing issues. I understand that not everyone has this freedom, but it's important to know that you can be picky if you choose to be.

Being picky doesn't only apply to projects. You can also be picky when it comes to the clients you work with. Working with individuals with a small budget is rarely worth your time. Not because these people aren't fun to work with, but because they have no experience in the space and have a limited budget to spend. Their focus is almost always on limiting costs and understandably so. The quality of the product is secondary. This means that the developer needs to work fast, cut corners, and almost always defend their decisions.

Taking a Peek

If you do decide to work on an existing project, I strongly recommend that you take a look at the project first. This usually isn't a problem. If it is, then that may be a red flag for you.

Taking a look at the project usually means signing a non-disclosure agreement. That's fine, but make sure you carefully read the agreement. You're only vetting the client and their project. Don't take any risks and don't commit to anything you don't feel comfortable with.

Is It Worth It

Instead of taking on the project, have a conversation with the client to get an idea of the amount of work they can bring in over the next months. Be careful, though. I have spoken with many potential clients that promised more work in the future and never delivered. That's why I never start a relationship with a client based on a project that only involves bug fixing and minor improvements. I understand that such an assignment is ideal for the client to test the waters. I see that a bit differently.

If a client approaches me for a collaboration, I expect them to trust me. They contact me for my expertise and I have no intention of convincing them that I'm a good fit for the job. If they feel I'm not, then they shouldn't have contacted me in the first place. Does this seem arrogant or complacent to you? Think again. I consider it a form of trust and respect.

Firing Clients

Most freelancers have the feeling that a client chooses them to work on a project. This is usually true if you don't have many projects lined up or if you're just starting out. Earlier I wrote about being picky and that also applies to clients.

As a freelancer, it's important to remember that you're on your own. No matter how good your relationship is with your client, they will let you go if they feel that's the best decision for their company. This means that you should always put yourself first, not your client. This applies to your health, your workload, and your mental sanity.

I've always been confused by the use of "freelancer" and "consultant". While there *is* a difference, a good freelancer is also a consultant. An experienced freelancer is worth its weight in gold and a client should be lucky to have you. With that attitude, your relationship with the client changes dramatically. You no longer feel that you should only do what you're told. You should also advise the client.

If you feel that your input and your advice are ignored by the client, then you may be starting to wonder why you chose to become a freelancer. When there's an issue with my car, I trust my mechanic to decide what needs to be done. A professional listens to the client and then decides what should be done. If the client doesn't respect your advice and insists that you do what they think is best, then you may want to fire the client. Wait. What? Fire the client?

Fire the Client

Firing a client is a new concept for many freelancers and subcontractors. It can be a scary thing to do, but if you feel that you're having an unhealthy relationship with your client, then consider taking action to change the situation. Having a conversation is obviously your first step, but, if that doesn't bring significant change, then you should consider moving on.

Part 7: Products

1 Ship, Ship, Ship

Steve Jobs famously said “Real artists ship.” He meant that putting your work out in the world is an essential aspect of every creative, and that includes developers. Creatives that only work in isolation and never ship anything may not be that *real* after all.

Earlier this year, I sent a survey to Cocoacasts readers to learn more about what they do, what their goals are, and where they’re currently at. I also asked them about any applications they have in the App Store. Many of us have published some personal projects, but the most common answer to the question was the goal or ambition to have an application in the App Store. It’s common to start something without ever finishing it.

It’s Hard

I’ve written about this many, many times because it’s something that fascinates me. Developers are creative people, and we have an urge to create. Unfortunately, most of us stop when we lose interest. Motivation isn’t what’s going to get you from idea to App Store. It’s grit, tenacity, and persistence.

Solving a problem is what we like to do. Writing release notes, implementing in-app purchases, or localizing an application isn’t what we enjoy most. But it’s an essential aspect of software development. Unfortunately, it’s what stops most of us from shipping.

How to Ship Consistently

Over the years, I’ve built many, many applications and some of them made their way to the App Store, not all. It has taught me several useful lessons that guarantee that I ship stuff consistently.

Make It Tiny

It’s fine to dream and to be audacious. But Rome wasn’t built in a day. What stops many of us from shipping our creations is aiming too high. I don’t mean to say that you *shouldn’t* aim high. What I *am* saying is that you should aim for consistency.

My latest application is focused and to the point. I have big goals, and even dreams, for this application, but the first release focuses on the essentials. I interviewed people that are interested in the product and asked them which features were essential for them to use the application. I analyzed the results, went to work, and plan to ship the first version of the application later this month.

It’s fine that your application is limited in scope. That’s much better than no application at all. Right? I’m convinced that there are thousands of applications collecting dust on people’s computers because they aimed too high for the first release. Keep it small and focused.

Ship Frequently

Once you have something in the App Store, it’s surprisingly easy to iterate on what you already have. Define the scope for the next release, keep it focused and small, and ship. Rinse and repeat. Listen to feedback and improve the application as you go.

It’s an approach that has worked very, very well for me. It also shows my customers that I’m continuing to invest in the product. It builds trust and confidence. The App Store is littered with abandoned applications that no longer receive updates. One of my favorite utilities stopped working after I updated my phone to iOS 11.

Keep the Bigger Picture in Mind

Shipping frequent, focused updates is very effective, but make sure you keep the bigger picture in mind along the way. Avoid ending up with a product that packs dozens of features but lacks a clear direction. You’re the product owner and it’s your task, not your customers’, to give direction.

Internal Deadlines

I sometimes make the mistake of publicly committing to a deadline. This strategy works for drawing attention, but it's detrimental for your sanity.

What works much better is setting internal deadlines, which I discuss in the chapter on source control. It works well in teams, but it's also an effective strategy for personal projects.

The idea is simple. Schedule releases on a regular basis and ship what's ready. That's the gist. If you adopt the branching strategy I describe elsewhere in this book, then that's simple to implement. Combine this with a dash of automation, and you have yourself a shipping machine. Such a workflow lets you focus on what you enjoy doing most, building amazing software.

Remove Friction and Clutter

There are numerous tasks you need to handle if you're building and shipping products. It can be a chore to prepare a release. To make this easier, I automate most of the trivial aspects of a release, such as translation management and creating and uploading screenshots. Manually creating and uploading screenshots is virtually impossible if you want to support every device your application can run on. You're a developer and you should be automating repetitive tasks.

It pays off to set aside a few hours from time to time and automate your workflows. I write more about this aspect of software development in the chapter on automation.

2 Talk to Your Customers

As developers, we have the itch or the need to analyze problems and craft solutions for them. That's what drives most developers. If you boil it down to its essentials, then a developer solves problems, large and small.

Some of us have bigger ambitions, though. What has always appealed to me most is the possibility of creating a product and selling it to hundreds or thousands of people. Apple gets a lot of flak for taking a significant cut from your earnings, but, credit where credit is due, Apple has made creating and distributing software easier than ever. It's no coincidence that the App Store catalog packs millions of applications. Whether that's a good thing is a different discussion.

Solve a Problem

There are many reasons for publishing an application on Apple's App Store. Some developers want to test the waters, create a portfolio to show future employers, and some of us want to help others for free.

Some of us have the ambition to turn our passion into a business. To be honest, I'm always suspicious when I see "passion" and "business" in the same sentence. I believe that you can be passionate about the thing that makes you money, but passion and business can be a dangerous or misleading combination.

Every business starts with an idea. But a successful business starts with a problem. Ignore Facebook and Twitter for now. I hope you don't have the ambition to create the next Facebook or Twitter.

As I mentioned earlier, a developer is wired to solve problems. Some of us also have a gift to spot problems, which can be both a curse and a blessing. But the first mistake developers make is trying to come up with an idea. And many people I've talked to get frustrated in this phase.

The solution is surprisingly simple. Turn it around. Instead of looking for an idea, look for problems and try to come up with a solution. That's the seed for a successful business. Why is that?

As developers, we're focused on the technical aspects of the product. We're so immersed in the process of solving the problem that we forget to ask the most important question. Does anyone want what we're building? And this leads to a waterfall of other equally important questions. Who is the person we're helping? What is the pain they have? And how are we trying to make that pain go away?

Ground Zero

Your first task is to validate your idea. It determines whether you will spend the next days, weeks, or months creating a business or a fantasy.

Every developer I've talked to about creating a product business has made this mistake. Every. Single. One. I've made this mistake several times, even after I knew I had to validate the idea first. It's just incredibly tempting to fire up a code editor and start coding.

Is This for You

If your passion is writing code, then building a product business may not be for you. You have to be honest with yourself about this. Building a business also includes marketing, customer support, writing release notes, and much more. You don't need to be passionate about these responsibilities, but you need to take care of them. I don't like marketing, but I know it's essential for my business.

The same applies to freelancing and working as an employee. Freelancing looks amazing on paper, but a freelancer needs to find clients, write proposals, and a lot of administration. That's why most developers work as employees at a company where they can focus on what they enjoy doing most. And that's fine.

But It's Fantastic

Even though running a product business is hard work, and a lot of it, the feeling of creating something people are willing to pay for is amazing.

You have the feeling that you're making a dent in the universe, a tiny one, but it's still a dent. You're doing something that matters and that helps people in their lives.

And that too can turn into a passion. Being passionate about your business sounds much healthier than trying to turn your passion into a business.

A Recipe for Success

While building a successful business is hard work, there is an easy blueprint you can use. It doesn't require venture capital, and you don't need to have thousands of followers on Twitter. The recipe is simple.

- Step 1: Find a burning problem people have.
- Step 2: Solve that problem.
- Step 3: Ask for money.

This roadmap isn't rocket science. Unfortunately, the path most developers that turn entrepreneur take is the following.

- Step 1: Come up with an idea, not a problem.
- Step 2: Build a product, not a solution.
- Step 3: Release it.
- Step 4: Frantically look for potential customers.
- Step 5: …

Like most freelancers, people occasionally contact me to talk about an idea they have for an application. They're prepared to spend thousands of dollars of their savings into a product that hasn't even been validated yet. They haven't talked to a single potential customer. This is a recipe for disaster.

As a rule, I don't take on projects I don't believe in. It isn't fun to see someone pour their savings into a project that isn't going anywhere. Don't make this mistake yourself. Talk to your customers.

3 What Is Stopping You From Shipping

Earlier in the book, I wrote about building and shipping something, putting something into people's hands. I understand that life gets in the way from time to time, but I'm genuinely wondering if you're creating something.

I'm not referring to the work you do for your employer or your clients. What I'm speaking about is the creative process of finding an itch, coming up with a solution, and creating a product that scratches that itch.

Your product doesn't need to change the world. You can start small. But you have to start. What are you working on? What itch are you scratching?

Why Is This Important

Building and shipping a product is something I write and talk about frequently because I believe it's important for every Cocoa or Swift developer to be familiar with the life cycle of a product, from start to finish. It's true that you're a developer, but that doesn't mean you should ignore other aspects of product development.

In medium to large companies, people often have the luxury to focus on what they do best and enjoy doing most. But I don't see this as a luxury. I firmly believe that a key strength of most freelancers and independent developers is their deep understanding of what it means to ship a product, from idea to App Store.

> I firmly believe that a key strength of most freelancers and independent developers is their deep understanding of what it means to ship a product, from idea to App Store.

It can be tough and frustrating to have to manage every aspect of the product's life cycle, but it's also interesting. It teaches you a lot about your craft, and you develop skills that make you an attractive hire. If you

don’t have the opportunity to be more involved in product development at the company you work, then you can, and should, create a product of your own.

Start Small

Having big dreams is great. It’s admirable. We need people with vision and ambition. But everything starts small. Thinking big isn’t a problem as long as it doesn’t distract you from the work that needs to be done today. Big goals always start small.

> Big goals always start small.

Start small by focusing on that one feature that sets your application apart from every other application. How can you make it stand out? What can you do to bring value to the user of your application with that one feature? Not two. Not three. Only that one feature.

By intensely focusing on one piece of functionality, you force yourself to make that one feature great, to make it magical. It’s fine to add bells and whistles, but they shouldn’t distract the user from center stage.

Apple’s original iPod put the company back on the map. It was a revolutionary product. It’s true that it looked amazing and that it was easy to use. But, at its core, it did one thing very, very well. Playing music. The product was focused. Apple removed every bit of clutter that could potentially distract the user. Focus is what made the iPod stand out from the rest of the market.

Remove Clutter

Many projects have a list of features that goes on and on. And that’s fine. But is the feature list of your project stopping you from shipping? Are you delaying the launch of your project because of one or two features that your application can probably do without?

If your project needs a dozen or more features to add value, it may be time to go back to the drawing board. If you need the bells and whistles to cover up that center stage isn’t that great, you have missed the point.

Too many features is all too often the cause of missed deadlines and a product that isn't being shipped. I'm currently wrapping up a brand new application. Some of the people testing the application are asking for an Apple Watch application. I plan to support Apple Watch, but the first version won't include this feature. I know it will take me several weeks or months to add this feature and that's not what I have in mind. I want to have this application in the App Store before the end of this month.

By postponing support for Apple Watch I make sure I can ship the application sooner, but it also allows me to focus. Focus is a word you find very often in this book. It has become an essential ingredient to success in today's day and age. I know it has for me.

What is stopping you from shipping?

4 Motivation Will Get You Only Halfway

I bet that you started more projects than you can remember. How many of those projects have you finished? How many applications did end up in the hands of customers?

You've probably heard the phrase "Follow your passion." or "Do what you enjoy doing." I agree that you should do what you enjoy doing, but you also need to know that passion often isn't enough to go from idea to product, a product people can use and find value in.

Motivation Won't Cut It

As developers, we tend to launch a code editor whenever we have a brilliant idea, coding away to solve the problem at hand. This impulse usually lasts for a few days or weeks. If you're lucky, you can motivate yourself until you have a working prototype you can share with friends or family.

A prototype isn't enough, though. Building a product people can use requires much more than a rudimentary version that solves a problem.

Running a Marathon

Perseverance, tenacity, and grit are qualities that are often undervalued or overlooked. Your boss or clients don't pay you because you're passionate or motivated. They pay you because you convinced them that you can get the job done.

Building a product, shipping it to customers, adding features, and fixing bugs are essential components of creating a successful product. Few developers get excited by bug or crash reports. The truth is that your code contains bugs and your application crashes if it has any complexity to it. Even though fixing bugs and investigating crash reports are less enjoyable tasks, they're integral aspects of software development.

> Creating a product isn’t a sprint. It’s a marathon with many ups and downs.

If you’re waiting for motivation to kick in before you start working on those bug reports, you’re probably waiting in vain. It requires perseverance to take a project from idea to shippable product. It takes tenacity to iterate on a product, to fix bugs, and add features.

Creating a product isn’t a sprint. It’s a marathon with many ups and downs. It’s fine to start with a sprint, but you need to remember that you’re in it for the long run.

Pulling the Plug

Not every product is going to be a success. The App Store is littered with failed and abandoned projects. Not too long ago, Apple and Google used the number of applications in their mobile stores to market their platforms. Sadly, this is no longer something to brag about. Apple is actively removing applications that no longer comply with the App Store guidelines. Apple has realized it’s time to focus on quality instead of quantity.

Even though Apple’s App Store contains millions of applications, finding what you’re looking for is difficult. Finding an application that’s frequently updated and fits your needs is even more challenging.

It’s fine to pull the plug on a product if you believe it’s no longer worth pursuing as long as it isn’t an excuse to work on the next brilliant idea that enters your mind. This is commonly referred to as the shiny object syndrome.

Start and Persevere

I’ve written about building and shipping products quite a few times in this book because it’s something I frequently struggle with. Adding features is nice, but updating translations, making screenshots, tracking bugs, and responding to customer feedback is much less glamourous.

Building and maintaining a product is more than worth it, though. Samsara, for example, has been around for several years and people still

email to tell me how much they enjoy using it for their yoga and meditation practice. These emails get me excited to continue improving Samsara with every release. It gets you through the tough times when motivation is low.

Challenge Yourself

Most developers thrive when they’re faced with a problem they need to solve. Continue to challenge yourself. Pick up a new language or explore a new library to keep you sharp. Solving problems is exactly what software development is about. Is it not?

5 How to Make a Living as a Mobile Developer

Developers often refer to the early days of Apple's App Store as the **gold rush** of the App Store. Paid applications were the norm and prices were much higher than they are now. What options do you have as a developer or entrepreneur in today's App Store? Is it possible to make a living as an independent developer? In this chapter, I list the options you have to make money from your applications in Apple's App Store.

Paid Up Front

The most common approach to make money on Apple's App Store used to be asking users for money. Really. It sounds too crazy to be true. Asking for money in exchange for a product is crazy. Joking aside, from a business perspective, this makes perfect sense, and it comes with virtually no overhead from the developer's perspective.

To make money in Apple's App Store, paid applications were the norm for the first few years. This period ended with what's now known as the **race to the bottom**, developers competing with each other by cutting the prices of their applications. The race to the bottom forced developers and businesses to find other ways to make money.

While I've played with Samsara's pricing, it has always been paid, and it has done pretty well compared to the majority of applications out there. While paid applications still have a future, it's a less popular option these days.

Freemium

The freemium model predates the mobile era. Desktop and web applications used the freemium model long before the iPhone was introduced. The idea is simple. You offer a product for free, getting it in the hands of as many people as possible. To generate revenue, you do your best to convince a subset of your users to pay for a premium experience.

Offering the product for free has two important benefits. Your product spreads faster and more easily, and you have a foothold in the customer's life.

How you define a **premium experience** depends on the product. Evernote is a fine example of a company that has successfully applied the freemium model. The company has tens of millions of customers. A fraction of those customers pays for Evernote's premium services. The company would have had a harder time gaining market share if it had offered its suite of products paid up front or with a monthly subscription.

Freemium isn't without risk, though. For years, Rob Walling has been warning developers and entrepreneurs that freemium is difficult to pull off, especially for small, bootstrapped companies.

If your product has a backend, for example, then keep in mind that customers that don't pay for your product also make use of that backend. Hosting costs can skyrocket if your product gains traction. In that case, the conversion rate of free to paid customers will determine whether your business is viable in the long run.

Advertising

Another widespread business strategy is advertising. You offer your application for free and display ads to your customers. This business model only works if your application has lots and lots of users and if your users frequently use your application.

Games are a good fit for ads. Flappy Bird showed that developers can potentially make millions of dollars through advertising as long as the application has enough users that frequently play the game. Flappy Bird was incredibly addictive, and that was the key to its short success.

David Smith is a successful, independent developer and he published a very interesting article about how his business has evolved over the years. In the early days of the App Store, the majority of David's revenue came from paid applications. Nowadays, advertising brings in most of the revenue.

> The market has been pulling me along towards advertising based apps, and I've found that the less I fight back with anachronistic ideas about how software "should" be sold, the more sustainable a business I have. — David Smith

For Overcast, Marco Arment has experimented with several business models, including advertising. At the time of writing, Overcast displays ads, which you can remove by subscribing to Overcast Premium. The ads shown in Overcast are managed by Marco himself and are primarily targeted at podcasts. He recently mentioned on ATP that this is working very well for him. It's interesting to see this change and it shows that you can make changes to the business model of your products as the space you're in evolves.

In-App Purchases

Selling virtual items is a very common monetization strategy in mobile games. Most of the games that top the charts of the App Store make money through in-app purchases. It's a bit ironic to see a free game at the top of the **Top Grossing** chart, but that's the reality.

You can use in-app purchases for many purposes. Some applications are free to download and offer premium features through in-app purchases. This approach is very similar to the freemium model we discussed earlier. Other applications display advertising, offering the option to remove ads through an in-app purchase.

Whether in-app purchases work for your product depends on your value proposition and it requires a bit of experimentation. If you only show a tiny ad in the settings view of your application and you offer an in-app purchase to remove it, then the chances are that people are fine seeing the ad every now and then. Don't discard in-app purchases if you feel it isn't working. It may take a few tries to get it right.

Subscriptions

Few mobile applications make money through subscriptions. If your application offers a service that continues to be of value to the customer, then offering a subscription makes sense.

Subscriptions are the holy grail in my opinion. They are great for building a sustainable business. Instead of starting from zero at the start of every month, you build up **MRR** or **monthly recurring revenue**. With a predictable, recurring revenue stream, building a business becomes a bit less risky.

If that's true, why aren't subscriptions more common in the App Store? Netflix offers subscriptions because its customers watch movies and shows on a daily or weekly basis. The company adds new movies and shows almost daily.

Most mobile applications can't offer a similar value proposition. A camera application, for example, isn't a good fit for a subscription model. Customers expect to buy a camera application once and use it as often as they want.

The landscape is changing, though. Last year, Apple announced that auto-renewable subscriptions are now available for most applications in the App Store. Also, after the first year, the developer's cut increases from the usual 70% to 85%. It seems Apple is encouraging developers to experiment with new business models, including subscriptions. It shows that Apple understands that the market is changing and evolving.

Donations

A small group of applications generates revenue through donations. With the release of Overcast 2, Marco Arment made its popular podcast client free. He introduced a patronage model in Overcast 2 to generate revenue from his product. To the surprise of many skeptics, patronage seemed to work for Overcast. But Marco wasn't entirely happy with the low conversion rate and he started experimenting with advertising and auto-renewable subscriptions.

Marco's reasoning makes perfect sense. He wants to offer the best user experience to everyone that chooses for Overcast. He doesn't want to offer an inferior user experience to people that don't pay for the premium features or, more importantly, to those that haven't decided yet whether Overcast is right for them. It makes sense, but I wonder how viable patronage or donations are for niche applications or applications with a smaller audience.

I've considered this approach in the past for Samsara. A patronage model would be a good fit, offering the best possible user experience to every user. From a developer's perspective, it's a compelling idea.

Finding the Right Business Model

Every application is different. Not every business model is a good fit for an application. If you're creating a camera application, then offering a subscription is probably not going to work. Using in-app purchases to unlock premium filters is a more viable business strategy.

It's worth spending time considering your options and experimenting with different business models. It's fine and possible to make your application paid up front and switch to freemium down the road.

> It felt hopeless, but my initial thinking was restricted by trying to wedge traditional software business models into the realities of today's App Store. It's hard to make older revenue models work today because the market is completely different. — Marco Arment

Experimenting is fine, but make sure you don't burn the goodwill of existing customers. If you switch from paid to freemium and force your loyal customers to unlock premium features they already paid for, then prepare yourself for a flood of angry complaints.

Experiment But Be Smart

Always keep your customers in mind. Don't switch business models with every major release. Does that mean you can't make changes to the business model of your product? Absolutely not. Let me give you an example.

Let's assume you release the first version of your application as a paid application. After a few months, you realize it isn't doing as well as it could. You want to make the application free with an in-app purchase to unlock premium content.

You could simply implement the in-app purchase, but this would anger customers that purchased your application when it was paid up front. The

solution is simple, but it requires some work. The App Store receipt includes the original purchase date. Your application can inspect the receipt and only offer the in-app purchase to customers that downloaded the application after you made the switch from paid to free with an in-app purchase.

Evolution

The mobile space continues to evolve and that also means that you and your business need to evolve. This doesn’t need to be scary, but you need to be vigilant. Keep an eye on the industry. Which trends are emerging? What’s working for other developers? Talk to your customers? What are their expectations? You may be surprised by their responses.

PART 8: YOU

1 Being and Staying Productive

Productivity is a topic that many people have written about and I'm certainly not a guru when it comes to productivity. Through trial and error, I've learned what works and what doesn't.

Productivity is a very personal subject. What works for me may not work for you. For that reason, I won't be bombarding you with tips and tricks in this chapter. Instead, I cover a few key elements that directly and indirectly impact your productivity.

Maintaining Focus

One of the biggest challenges we're faced with as developers is maintaining focus throughout the day. The people we work with often expect us to be available at all times. If you cannot cope with that feeling, it can add a substantial amount stress to your life.

That's why it's vital to set boundaries. This is even more true for freelancers and consultants. If you run a business, people seem to expect you're working every waking moment. You *are* the business and you should always be available.

Don't make that mistake. Set boundaries for yourself and for the people you work with. Believe me. People will respect you for it. And those that don't may not be the people you should be working with.

Protect Your Productivity

Open offices are very popular in the technology industry. While I understand their appeal, I'm not a fan. It's true that they promote communication and collaboration, but they're almost always destroyers of productivity. Because your desk isn't surrounded by four walls, people tend to think that they can step into your imaginary office whenever they feel like.

As a developer, you need focus and you need to know that people won't disturb you when you're writing code. Most of us know this, which is why open offices are usually filled with people wearing headphones. The headphones are a silent protest against the lack of an office or a space that allows people to focus on what they're supposed to do.

Make it clear to the people you work with that they can't or shouldn't disturb you whenever they like. Isn't that what the daily scrum meetings are for?

Distraction Is Addictive

As if open offices aren't enough, instant messaging, like Slack, are used in almost every technology company. They're incredibly useful and, at the same time, a major distraction for most developers. The same is true for instant messaging of all forms and shapes.

Being distracted has become an addiction for some of us. We get a kick every time we see an email come in, a message on Slack, or a new tweet on Twitter.

For many of us, it's a problem, a real one. We're being distracted and we're craving for distraction. I don't have the perfect cure to battle distraction, but I have a piece of paper taped to the wall opposite my desk with three lines of text.

> Have Focus, Remove Noise, Set Boundaries

Have Focus

This is a very effective strategy if you stick to it. The first line is simple. Make sure that you know what you're doing. Your focus should be singular. This may sound obvious, but many of us start to procrastinate and look for a distraction when we don't have a clear focus. Every time I don't have a clear focus I ask myself a simple question.

> What's the next action I need to take to move forward?

I believe I read this in a book by David Allen years ago. It's a powerful technique to get unstuck and move on. It works very, very well if you're battling a complex programming problem, because, let's be honest, even complex problems and solutions can be broken down into simple, manageable pieces.

If you're having a rough day, I recommend taking tiny steps. Launch Xcode. Open the project you're working on.

Remove Noise

Focus is only possible if you remove anything that can distract you. Set a timer for 30 to 60 minutes and remove every possible source of distraction. I'm writing this on a laptop without an internet connection in a dimmed room on an improvised standing desk. I have notifications disabled, closed browser windows, and I can only be disturbed by emergencies.

Is this overkill? I don't think so. How do you think programmers got work done twenty or thirty years ago? They weren't distracted by Slack, mobile phones, and a constant influx of news, tweets, and messages. We haven't evolved since those days, which means we are not immune to the distractions of today's world.

I have one pro tip for you. If you're working on an iOS application, make sure you use a test device, not your personal device. It makes it easier to block out messages, phone calls, and other distractions that your phone receives.

Set Boundaries

This is a big one. A few months ago, I read an article on productivity by Ramit Sethi. In the comments of the article, a reader asked Ramit about being overwhelmed by work and the feeling that everything was getting too much. Ramit responded to this person's comment and pointed out that overwhelm is almost always caused by a lack of boundaries.

I wrote about setting boundaries earlier in this chapter. Not setting boundaries can mentally and physically exhaust you. How do you set boundaries? Turn off your computer at 6PM and make it a habit that you

don’t check email or messages from work after that time. That’s a good start.

This can be challenging if you’re not used to do this, but after a few days you should feel more relaxed and more energized the next day. A spring that’s constantly stressed loses its flexibility and the same happens to people. The result can be burnout or depression, both of which are difficult to recover from. This brings me to the last section of this chapter.

Take Care of Yourself

A runner running a marathon doesn’t run all out for the entire race. She keeps something in the tank for the finale of the race. Working as a developer is similar. Make sure you don’t push yourself to your limits day in day out. Make sure you have a healthy reserve for when it gets tough or when things hit the fan.

You’re in it for the long run and that means that you need to take care of yourself, including your body. I’m in my thirties and burning the midnight oil is something I can no longer pull off without suffering the consequences. I see young people drinking energy drinks and bucketloads of coffee. This isn’t healthy and you will pay the price at some point. You probably won’t believe me if you’re in your twenties.

2 Stop Looking for the Silver Bullet

Most frameworks and libraries have guidelines and best practices. Apple's SDKs are no different. While there's room for experimentation and exploration, a typical iOS application, for example, uses one or more view controllers and the user can navigate between those view controllers. That's a recipe you're probably familiar with.

Not every problem you face during development has one solution, though. This is a common issue developers face when they use Core Data, for example. They run into an issue and they try to resolve it by looking for the recipe that fixes their problem.

An Example

A few years ago, Florian Kugler wrote an excellent article in which he compares three Core Data stack configurations. In his article, Florian tests how each Core Data stack performs when the application imports a large data set in the background. The results were surprising, to say the least.

The Core Data stack Apple and several known experts recommend performed terribly. I was surprised by the results and so was Florian. He investigates the test results and unravels why a Core Data stack with a parent-child configuration doesn't handle large background imports very well. It's a very good read if you're interested in Core Data performance.

To resolve the issue, Florian didn't go looking for the silver bullet in the Core Data programming guide because there is no silver bullet. There are best practices and guidelines, but not every problem can be solved with a clear-cut recipe.

Investigate and Test

A few months ago, I came across an article by Sam Soffes about improving application performance with the help of **Instruments**, one of

the most underused developer tools for Cocoa development. In his article, Sam describes how he spent an afternoon investigating three performance issues in an application he was working on.

As with many performance issues, the bottlenecks were unexpected. One issue, for example, was caused by a date parser that parsed ISO 8601 dates. He only discovered this issue by profiling the application's performance.

Sam tested, investigated, and improved the application's code base. He didn't follow a recipe and he certainly didn't search for a silver bullet to solve his performance problems.

Stop Looking for the Silver Bullet

If you're new to a framework or library, it can be frustrating or even scary to know that you're on your own to fix performance issues or other issues that are specific to your application. But that's the reality of software development.

It's true that you may be using the framework or library incorrectly. That's actually a common cause of bugs and performance problems. The moment your project gains in complexity, you're often on your own. You can learn from other developers that have faced similar issues, but it's important to stop looking for the silver bullet that you may think you can find somewhere.

Instead, profile your application, investigate the results, and try to improve performance problems or fix that hard to find memory leak. That's how you grow as a developer. Being afraid of exploring uncharted waters is understandable, but that doesn't mean you shouldn't give it a try.

3 You Are Not an Imposter

Have you ever heard of imposter syndrome? Like Pauline Rose Clance, I prefer imposter experience as it better describes the problem.

A surprising number of people suffer from imposter experience. And developers are no exception. What is imposter experience? And why am I writing about it in this book?

What Is It

People suffering from imposter experience are generally very good at what they do. They're often perfectionists, giving their very best. Yet they feel insecure and, at times, they think of themselves as frauds or imposters. They feel as if they're tricking the world into believing that they're better than they actually are. Does that sound familiar?

If you want to learn more about imposter experience, I recommend reading the Wikipedia entry on the topic. It does a great job at explaining the details.

Why Do I Bring This Up

Regular readers of Cocoacasts know that I occasionally write about the psychological aspects of being a developer, working as a freelancer, or running a business. These are topics I have a special interest in since they're an integral part of being a developer.

Topics like imposter syndrome or experience are often ignored, discarded with a laugh, or trivialized. But if you're a developer and you want to live a happy life, enjoying the work you do isn't enough. It's equally important to feel comfortable around the people you work with and the environment you spend a good chunk of your time in.

Curing Imposter Experience

It's important to emphasize that I'm not a psychologist and have no medical training whatsoever. The advice I offer is based on my own experience and conversations with developers like yourself.

Honesty

Imposter experience is pretty common among freelance developers, especially if you're working in a team of one. Why would anyone pay you money for the code you write? And what if people find out that you're not as good as they thought you were?

One of the best cures for imposter experience is honesty and openness. When Swift was introduced in 2014, apart from a handful of developers at Apple, nobody knew what Swift was. Best practices were still waiting to be discovered and everyone played with and explored the language.

Natasha Murashev has been a true inspiration for many developers. From the start, Natasha has openly documented her exploration of the language, sharing her discoveries along the way. Even though she's a fantastic developer, she doesn't proclaim to be an expert. Especially in the early days of Swift, her blog read like a diary of how she explored the language. She studied Swift honestly and openly, the perfect remedy to cure and even prevent imposter experience. It makes Natasha's blog fun to read and it immediately creates a human connection.

Don't Make Promises You Can't Keep

Years and years ago, I taught students at a university in Belgium. One of the key lessons I learned from one of my mentors was to be up front with your students about what you know and, more importantly, what you don't know. If a student asks you a question you don't know the answer to, tell them.

Admitting that you don't know something makes you human. You're not a robot and your brain isn't plugged into Google—not yet anyway. But there's more. Most people show respect if you can admit that you don't know or cannot do something. What can seem like a sign of weakness is actually a sign of strength and confidence.

It Automatically Disappears

I don’t feel it’s important to understand the root cause of imposter experience to rid yourself of it. After a while, it disappears automatically. Whenever I discuss a project with a client, I’m open and honest about what’s possible and what isn’t. If I’m not familiar with a specific technology, I tell them and provide an alternative.

Being open increases your credibility and it strengthens the relationship with the client. It’s a powerful antidote for imposter experience.

Talk About It

I’ve been wanting to write this post for quite a while. What inspired me to take action was a recent episode of the Tropical MBA podcast, TMBA384: Are You an Imposter?. One of the key takeaways from the episode is simple, surround yourself with like-minded people and talk about it.

> You only feel like an imposter if you compare yourself with others. — Luis Miguel Gil

Dan interviews several people that openly talk about imposter experience. Luis Miguel Gil opens up about how he experienced imposter experience. He also zooms in on what causes imposter experience and how to defuse it. This is what Luis has to say about feeling like an imposter.

> When I compare myself with Luis from three years ago I see huge growth, huge success. Amazing. I’m very happy and very proud of what I’ve accomplished. But when I compare myself with people like you guys (Dan and Ian, the hosts of the podcast) or other entrepreneurs that have had many amazing successes, I feel I’m far away from reaching that. I’m so far away. — Luis Miguel Gil

It can sometimes be easy to become overwhelmed or intimidated by what others are doing and have achieved, especially if you watch too many talks or read too many blog posts. It’s important to appreciate yourself and don’t undervalue what you’ve already achieved.

It’s Common

Imposter experience isn’t uncommon. Many people suffer from imposter experience, including people that are considered top performers in their field. Tom Hanks, one of my favorite actors, made a surprising comment several years ago.

> I still feel sometimes that I’d like to be as good as so-and-so actor. I see some other actors’ work, and I think I’ll never get there. I wish I could. — The New York Times

This may seem odd for someone who’s won two academy awards. It shows that it isn’t uncommon. It makes him human and approachable.

Don’t Believe Everything You Read

Imposter experience can also be fueled by believing everything you read. Some people may claim that they built a successful business in two months by working five hours a week. That may be possible, but they’re the exception to the rule.

Don’t compare yourself with exceptions. If you want to achieve something that you’re proud of, you need to put in the work. If you want to become a great developer, you’ll need to write code, and lots of it. And spending your days reading tutorials or browsing Stack Overflow won’t help much.

Don’t Let It Affect You

Most developers suffer from imposter experience at some point in their life. And that’s fine. It won’t kill you. But Dan Andrews emphasizes that impostor experience can grow into a problem if, how you run your business or your career, is affected by imposter experience.

This is a problem I see many freelancers struggle with. They undervalue what they have to offer. Something that seems easy or even trivial to you can be of great value to others. That doesn’t mean you need to undervalue yourself or, heaven forbid, do it for free. The reason you find it easy is because you’re an expert at what you do. And that’s why people want to pay you for your services and expertise.

A carpenter gets better the longer he does his job. This means he gets better the older he gets. Do you think he drops his hourly rate as he gets better and grows older? The opposite is true. Don’t think for a second that you’re any different.

Success Through Failure

Failure continues to be a touchy subject in many parts of the world. I feel it’s much less of a taboo in the United States. Let’s be honest, if you succeed without failure, then you haven’t aimed high enough.

Every single person that has reached success has experienced failure. It’s not something to be embarrassed about. You can only succeed by trying, failing, trying harder, failing again, and persisting until you succeed.

Remember this, if you succeed without failing, you’re not aiming high enough. Be realistic, but make sure you’re not underestimating what you can achieve. Some things are hard, but that doesn’t mean they’re impossible.

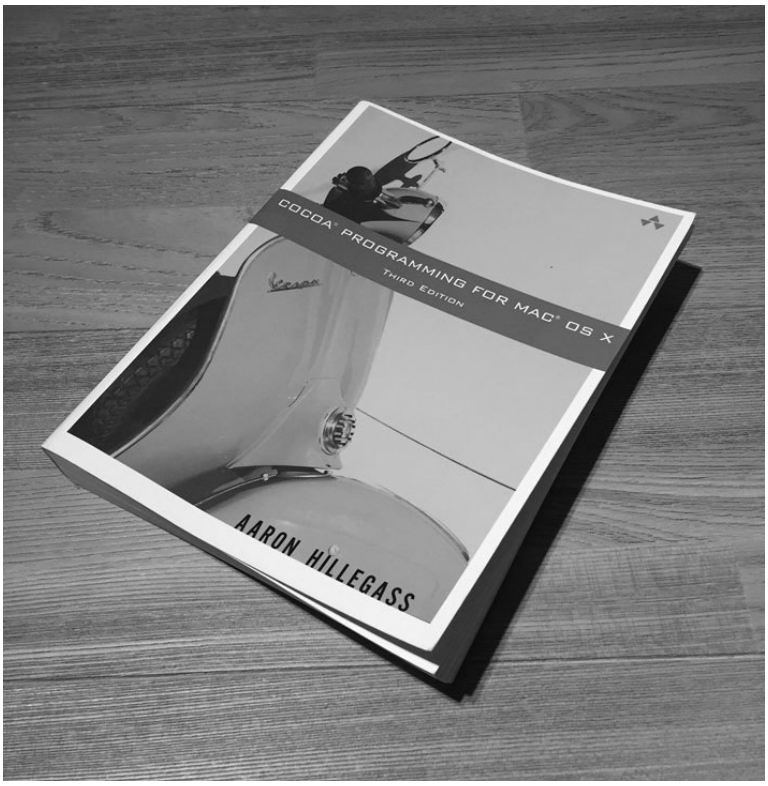
PART 9: LEARNING

1 Choose Your Teacher Wisely

The vast number of tutorials and courses about software development is a blessing for anyone interested in building software. And this is no less true for anyone interested in Swift development. Getting started with Swift development is easy and it doesn't need to cost you a fortune. But, as you've probably discovered, there's a downside to this wealth of information. In this chapter, I'd like to highlight three problems that I frequently face and hear about from my students and readers.

Information Overload

When I started learning Cocoa development in 2006, there were only a handful of books available. Aaron Hillegass' book Cocoa Programming for (Mac) OS X was the unofficial golden standard.



Cocoa Programming for (Mac) OS X

But, with the introduction of the iPhone in 2007 and the release of the official SDK in 2008, that number increased substantially. There are many books available from traditional publishers, such as Apress and O'Reilly, but many developers, including yours truly, have chosen the path of self-publishing, bypassing traditional publishers.

It has never been easier to publish a book or course. Books and courses are available at any price point, including free. Unfortunately, this has made it more challenging for developers to decide which path to choose to learn the topic they're interested in. Each day new tutorials, videos, and courses are published. It's challenging. That's for sure.

The blessing of having so much free information at your fingertips is more often than not a curse for developers that want to learn Cocoa and Swift development. Not only is the amount of information overwhelming, there doesn't seem to be a clear path to follow. It's difficult to see the forest for the trees. Don't stop here, though. There's hope.

Who to Trust

Every developer that writes about Cocoa and Swift development does this with the best of intentions. They want to help people learn something new or solve a problem they're having. It's fantastic to see how many of us take the time to write a tutorial or produce a video to help others. It's one of the key ingredients of the thriving Cocoa and Swift communities.

Despite the author's good intentions, though, what they're teaching may be incorrect or ignore good practices. If you're new to a subject, then you may not be able to spot these mistakes. The title of this chapter is very telling. It's become more important than ever to choose your teacher wisely. While I don't believe that people are intentionally putting out bad content or incorrect information, it's up to you, the student, to filter the good from the bad.

A few weeks ago, I was looking for a solution to customize the color of the clear button of a `UITextField` instance. It turns out that the tint color of a `UITextField` doesn't affect the clear button. Bummer. I was having an issue where the clear button was nearly invisible against a dark background.

During my search for an answer, I stumbled on several Stack Overflow entries that recommended digging into the view hierarchy of the text field, looking for the clear button, and modifying its tint color. As I mention elsewhere in this book, this is a bad practice. If the API doesn't allow for this type of customization, you file a bug with Apple and implement a custom solution. That's the only correct solution. Your clever workaround will inevitably break when Apple makes changes to the internals of the `UITextField` class. Respect the SDK. Always.

The answers that suggested digging into the view hierarchy of the text field were upvoted because, at that time, it solved the problem. Unfortunately, this creates a bigger problem. Every inexperienced developer that reads one of these answers is made to believe that this is a viable solution, that this is fine. Instead of spotting the risk of the solution, they wrongly believe that they've picked up a good practice they can adopt in other, similar situations.

If I need to pick up a new framework or API, I rely on the people I have come to trust over time. These are very often people that have built up authority in the community over several years or people like Aaron Hillegass, Marcus Zarra, and Jeff LaMarche that have been around since the very early days of the platform.

Focus, Focus, Focus

I used to follow a slew of developers on social media. I wanted to know what they were learning, what they had to share, and which techniques and lessons I could adopt in my own projects. About a year ago, I stopped doing this because it was too overwhelming. There's so much to learn. The platforms and the Swift language evolve so quickly that it's hard to keep your focus if you don't protect it ferociously.

There are countless talks about almost any topic you can imagine, but it can leave you with more questions than you started with. I don't know about you, but focus has been the cure for me. Attention has become so important in today's busy world that having the skill to focus obsessively is a skill every developer should learn to master.

I don't believe in the concept of a genius. It's true that some people are more gifted than others, but, in the end, developers that excel in what

they do are those who can focus and commit to something. You don't master Swift by reading a few chapters in Apple's language guide. That's a good start, but it's a process that continues and never stops. That's the beauty of it. No? Didn't you become a developer because the sky's the limit?

That's also why I often ask developers what goals they have for the coming months or years. What I'm actually asking them is what their focus is. What are they trying to accomplish? Ambitious people have a clear focus, very often a singular one. Let me ask you then, "What is your focus?"

Never Stop Learning

The mobile space is still very young, relatively speaking, and it evolves at an incredible pace. With Apple and Google heavily investing in their platforms, the speed with which mobile platforms evolve requires developers to focus and learn non-stop.

In 2015, Apple introduced no less than two new platforms, tvOS and watchOS. While developers familiar with iOS won't have a hard time getting up to speed with Apple's brand new SDKs, there are many, many APIs and paradigms to become familiar with. It's understandable if you feel a little overwhelmed as a mobile developer. That's fine and it's fine to admit that.

If you decide to become a developer, regardless of the platform you write software for, you need to accept and become comfortable with the fact that learning on a daily basis is part of the job.

For the past few years, Apple has released a new version of its operating systems every year, introducing new technologies we need to become familiar with. The Swift project continues to evolve at a fast pace. At the time of writing, Swift 4 is just around the corner and with it come new features, bug fixes, and numerous improvements.

Are you ready to dive in head first? If Apple announces a slew of new APIs next year, will that scare you or will it excite you? Do you have the motivation to not only continue learning but also push the envelope.

Using an API is one thing, pushing it to its boundaries and beyond is where it’s at.

If you like programming, but prefer to stick with what you know, then mobile development may not be the best choice. If learning is in your blood and the mere thought of WWDC or Google I/O gives you goosebumps, then being a mobile developer is the best job in the world.

2 Taking a Shortcut

What I like about programming and software development is that everyone is treated the same. This means that there are no shortcuts you can take. This is something I bring up several times in this book because I strongly believe that you need to put in the work if you want to evolve into a proficient Swift developer.

If your goal is to become one of the world's top violinists, then you simply need to put in the hours. The same applies to programming and software development. Why would it be any different?

Even though there are no shortcuts, there are several paths you can take to speed up your career. The shortcuts I mention in this chapter are legitimate recipes to accelerate your growth as a developer.

Internship

An internship at the right company can speed up your career dramatically. Why is that? A junior developer has a lot to learn and that's something many, many developers are overwhelmed by. What do I need to learn next? What do I need to know to build my first application?

There's a lot to learn and a lot that you could learn. Which topic is the next step you need to take. Having a mentor or a senior developer by your side can make a world of difference. For this to work, though, it's important that you carefully choose the place you apply for an internship. It's fine to be picky. It's not because you have little experience that you have to accept what you can get. Do your research, ask around, and contact the companies that you think are a good fit for you and what you want to achieve.

Freelancing and Subcontracting

The life of a freelancer may seem like the perfect life to many developers. It's true that it comes with several perks. You can start work whenever

you like and you can choose your workplace. But it’s a tough life at times, especially if you’re just starting out.

For that exact reason, however, freelancing can be a shortcut in your career. You’re forced to learn a lot of stuff and figure things out on your own. You need to take care of a million things to keep your business afloat. You need to develop skills you didn’t know you ever needed, such as writing proposals, talking to clients, and working with subcontractors.

This isn’t for everyone, but if this sounds like a challenge to you, then freelancing can significantly speed up your career.

Subcontracting is similar to freelancing. The difference is that you usually work at your client’s company. It can sometimes feel as if you’re an employee if you’re subcontracting. The reason I consider subcontracting a shortcut is for exactly that reason.

As a subcontractor, you usually become a member of the team at the company you work for. You learn about the company’s culture, the tools they use, their workflow, and their automation. The result is that you learn a lot in a short amount of time. You see what works and what doesn’t.

I worked as a subcontractor for several years and it has taught me a lot about the various aspects of running a business, project management, programing, automation, and testing. The bigger the company, the more effective this shortcut is.

As a subcontractor, you switch companies from time to time and that only increases the amount you learn. Every company does things differently and that’s what you’re looking for. You can pick and choose what works and apply that. It’s one thing reading about something, but it’s something completely different to be immersed in it.

If you’re a freelancer and freedom is important to you, then subcontracting may not be what you’re looking for. Keep that in mind when you’re given this opportunity.

Learn, Learn, Learn

If you’re reading this book, then it probably means that you’ve chosen for a career in the mobile space or you want to build fantastic products that help thousands or millions of people. The technology industry is evolving at a breakneck pace and you need to stay up to date. You need to learn every single day.

That doesn’t necessarily mean that you need to spend an hour a day with your nose in a book. It’s enough to have an open mind. What does that mean? Over the years, I’ve developed a slew of habits and strategies to solve common programming problems. But I try to do things a bit differently every time I work on a project. That’s how I discovered the Model-View-ViewModel pattern. The view controllers in my projects were getting too heavy and it was time for something different.

I studied the pattern for several weeks, tried a few implementations, and settled with a solution I satisfied my needs and requirements. That’s how I usually pick up new frameworks and libraries. Have an open mind and try to do thins a bit differently every time you encounter a familiar problem.

Build, Build, Build

One of the main reasons I continue to invest time and energy in side projects is to learn and experiment. If you’re working for a large agency or a product company, you don’t necessarily have the luxury of working with the latest and greatest. Many companies postponed the transition to Swift for the first few years because Objective-C was much more reliable and it had a proven track record. But what’s stopping you from learning Swift today? What’s stopping you from playing with Core Data or Realm in your spare time?

If you’re lucky, then you work at a company that gives you the freedom to learn new things on the job. While this may seem like a generous gesture, your employer knows and understands that they need to invest in their employees and their education. The technology industry is a competitive arena and what worked yesterday may no longer work today or tomorrow.

Choose Wisely

I started this book with a strong focus on your foundation and I want to briefly revisit this topic. It’s tempting to focus on the shiny stuff companies like Apple and Google spoil us with. But remember that you make a living building applications that are powered by technologies that have been around for decades. Make sure you don’t neglect your foundation.

With a solid foundation it’s easier to pick up new technologies. Once you understand how the puzzle fits together, it’s easier to pick up new languages, frameworks, and libraries. Investing in your foundation should always be your main concern. It helps you build confidence and it ensures that you can make a living doing what you love doing, building software.

3 Some Things Are Hard

Before the iPhone was introduced in 2007, Cocoa Programming for Mac OS X by Aaron Hillegass was *the* book if you were interested in Cocoa development. It was Aaron's book that introduced me to Cocoa programming.

One of the most important lessons Aaron taught me had nothing to do with programming. In the first chapter of his book, Aaron writes about Rock, a former boss he had worked with.

> I used to have a boss named Rock. Rock had earned a degree in astrophysics from Call Tech and had never had a job in which he used his knowledge of the heavens. Once I asked him whether he regretted getting the degree. "Actually, my degree in astrophysics has proved to be very valuable," he said. "Some things in this world are just hard. When I am struggling with something, I sometimes think 'Damn, this is hard for me. I wonder if I am stupid,' and then I remember that I have a degree in astrophysics from Cal Tech; I must not be stupid." - Aaron Hillegass

Some things are hard. Unfortunately, admitting that some things are hard is some sort of taboo among programmers. You lose credibility as a programmer if you admit that you're struggling understanding something.

Cut Yourself Some Slack

If you're just starting out as a programmer or you've been developing for years and are now testing the waters with a new language or technology, then cut yourself some slack if you're not making as much progress as you'd hoped. Some things are hard and that's fine.

Generics, for example, are mischievous creatures if you're coming from C or Objective-C. It's fine if you need a bit more time to let this paradigm sink in. Currying is a concept even some experienced developers avoid

because it isn’t easy to grasp if you’re not used to functional programming concepts.

That’s fine. Unless you’re a genius, you will struggle at times. That’s what makes programming fun. Right? What would the challenge be if every day was a walk in the park. Why would you invest time and energy in learning something if it was easy.

> It’s not that I’m so smart, it’s just that I stay with problems longer. â€” Albert Einstein

Programming is about solving problems and that includes learning a new language, framework, or technology. Challenging yourself is important, but it’s equally important to give yourself a break from time to time.

Some things are hard and that’s fine.

4 Learn the Rules, Then Break Them

When you're starting out as a developer or you're learning a new language, framework, or library, you're often wondering what the best practices are and which rules you need to stick to. Most of us have gone through this phase when we learned the Swift language.

I often bring up Core Data because it's such a good example. Developers are very often frustrated with the framework and that's in part why the framework doesn't have a great reputation. I can assure you that the vast majority of problems is caused by not knowing about or ignoring the rules of the framework.

Examples

Views are supposed be dumb. A view should only know how to present the data it's given. This is a rule you shouldn't break. But I don't agree with that. Once you understand why this rule is important, it's fine to break it as long as you understand what the consequences are.

I sometimes break this rule when I create a complex table or collection view. If you notice that you're making it very hard on yourself to keep the view dumb and ignorant, then it may be better to look for an alternative. In such a scenario, I promote the table or collection view cell to a controller. I see each table or collection view cell as a view controller and its content view as the view controller's view. This can often reduce the complexity in the view controller that drives the table or collection view.

I know that I'm breaking a rule and I understand the impact of that decision. That understanding gives me the confidence I need to implement a solution I know works better.

Another common example involves the Model-View-ViewModel pattern. You may have read that importing UIKit in the view model is a code smell. I agree with this statement and understand why that is. The view model shouldn't know about the view it powers through the view

controller. If you import the UIKit framework, it indicates that you're doing something in the view model that relates to the view layer hence the code smell.

This is clear. But I sometimes break this rule for obvious reasons. What if I want to return a `UIImage` instance from the view model or I want to specify the accessory type of a table view cell? The second example is harder to justify, but returning a `UIImage` instance is a fine example. The `UIImage` class is defined in the UIKit framework and if you want to return a `UIImage` instance, you have no other option but to import the UIKit framework. You could stick to the rules by just returning the name for the image, but is that really a better solution? Isn't that simply a workaround to abide by a rule?

Growing as a Developer

There's a subtle but important difference between violating a rule and breaking a rule. Breaking a rule implies that you know about the rule and you should know about the consequences. That said, break rules sparingly. There's a reason why design patterns and best practices exist. It can sometimes help to play by the rules to understand how to break them.

> There's a subtle but important difference between violating a rule and breaking a rule. Breaking a rule implies that you know about the rule and you should know about the consequences.

If you're new to a language or framework, then I recommend playing by the rules. This is essential to understand the language or framework, learn about the API, and how the various pieces of the puzzle fit together. You cannot safely break a rule if you don't know what the consequences are. Junior developers do this frequently without realizing it. And that's fine. You always learn something if you make a mistake. Just make sure you don't make the same mistake twice.

Creating Something Better

The concept of breaking rules is quite common in design. Revolutionary designs don't emerge by sticking to the rules. Even though this is less

obvious in software development, breaking rules can lead to better design patterns or practices. But it requires a deep understanding of those rules to pull it off.