Christian Tietze

Exploring Mac App Development Strategies

Patterns & Best Practices for Clean Software Architecture on the Mac with Swift 3 and Tests

Exploring Mac App Development Strategies

Patterns & Best Practices for Clean Software Architecture on the Mac with Swift 3 and Tests

Christian Tietze

This book is for sale at
http://leanpub.com/develop-mac-apps-clean-architecture-swift

This version was published on 2016-11-21

This book is for you.

It's for you because I think you deserve it.

You deserve to write clean code and make maintaining your app a joy.

I don't want you to waste any more time. Programming is hard enough as it is, no need to fight with the framework. That's why I wrote this book.

Thank you for caring.

– @ctietze

Contents

Prologue

Get the Latest from the Lab

Sign up to get a private preview of my upcoming programming books and other material to make you a better independent developer:

http://cleancocoa.com/newsletter/

Sample Information

Only a small selection of chapters are added to this chapter in this sample. I selected sections which I find most interesting to show you what is covered in the book. Because the sample is missing context, a few things might not make sense the way it is.

Who Is This Book For?

You have all the best intentions but no clue how to make them a reality: You want to write clean code and you don't want to end up with ugly and unmaintainable applications.

I was stuck in that same situation. I wanted to write my own apps and found my needs quickly surpassed the guidance of Apple's documentation and sample codes. Both are a useful resource, but you won't learn how to scale and create a complex application.

Typical questions include:

- How do you architect a complex application?

- How do you deal with large databases?
- How should you implement background services?

It's a popular advice that you should not stuff everything into your view controllers to avoid suffering from *massive view controller* syndrome. But nobody show how that works in apps. Maintainable code is important for an app that grows, so how do you get there?

I focus on laying the foundation for an answer to the first question, how to architect Mac apps. I won't tell you how to scale your database and how to split your app into multiple services in this book. (But check the website from time to time, because I've got something in the making!)

To read this book, you don't need to be a Cocoa wiz. You don't even need to be proficient with either Swift or Objective-C: I don't use crazy stuff like method swizzling or operator overloading or anything that's hard to grasp, really. When I wrote the first edition of this book in 2014, I just learned Swift myself. So there's lots of explanation and sense-making along the way. Chances are you already know Swift better required to understand the book.

Here's what I hope you will learn from this book:

- Your will learn to recognize the core of your application, the Domain, where the business rules belong. This is the very core of your application and it has no knowledge of UIKit, AppKit, whatever-kit – it's all yours, custom made.
- Consequently, you will learn how to put your view controllers on a diet. We will consider them a mere user interface detail.
- Along the way, you'll get comfortable with creating your own "POSO"s (Plain Old Swift Objects) that convey information and encapsulate behavior instead of desperately looking for canned solutions.

In short, the message of the book is: don't look for help writing your app. Learn how to create a solid software architecture *you* are comfortable with instead, and plug in components from third parties where necessary.

Contributing to the Book Manuscript and Example Project

I believe in empowering people. As a consequence, I also believe in sharing information. That's why I am open-sourcing the whole book plus example code. You can pay for the packaged product and support my writing, but the content will be freely available for every brave soul trying to make sense of Mac app development.

Your feedback is very welcome, and I appreciate pull requests on GitHub for both the manuscript and the example project: if you spot a typo or a code error, help me fix it! You can also open issues on GitHub to request additional "features:" more details, more explanations, more examples.

So feel free to contribute to this book and the overall project via GitHub:

- Book Manuscript on GitHub[1]
- Project Code on GitHub[2]

You can read more about the book project on its dedicated website[3].

A huge thanks for providing feedback and fixing bugs goes to:

- Jorge D Ortiz Fuentes
- Stanislaw Pankevich[4]

A Note on Reading the Example App Code

Developing an app is an iterative process. Looking at the code of a finished app often leaves you wondering: how did he get to this point?

Experimental code from the first commits won't be available in the final version, of course, because I've surpassed the problems along the way. I leverage the version

[1]https://github.com/CleanCocoa/mac-appdev-book
[2]https://github.com/CleanCocoa/mac-appdev-code
[3]http://cleancocoa.com/books/exploring-mac-app-dev/
[4]https://github.com/stanislaw

history of the example app to show the app at every stage of development so you can follow along. The git history becomes a tool to understand the development. Throughout the book, I include links to commits on GitHub so you can checkout the code at a particular stage in the narrative.

When you visit GitHub and download the code as a Zip file, though, you will see the final version. This final version will not reflect intermediate steps of the development process. Refer to the footnotes and the links to git commits if you find you need more context. I explain a lot of the refactorings and discuss most code changes in the book itself so you can follow along easily.

If you cannot understand what a section is about because there's not enough code to illustrate what's going on, simply open an issue on GitHub in the book manuscript repository and I'll revise this in a future edition!

Motivation to Write This Book

I develop apps[5]. My first big Mac application was the Word Counter. Back in 2014, I began to sell it. To actually make money from my own Mac software. And that made me nervous – because what if I introduce tons of bugs and can't fix them? How do I write high-quality code so I can maintain the app for the next years? These questions drove my initial research that led to the field of software architecture.

For an update in late 2014, I experimented with Core Data to store a new kind of data. That became quite a challenge: using Core Data the way of the documentation and sample codes results in a mess. Core Data gets intermingled everywhere. This little book and the accompanying project document my explorations and a possible solution to decouple Core Data from the rest of the app.

Adding a New Feature to the Word Counter

Rewind to fall 2014: I want to add a new feature to the Word Counter.

Until now, the Word Counter observed key presses and counted words written per app. This is indented to show your overall *productivity*. But it can't show *progress*, another essential metric. The ability to monitor files to track project progress was next on the roadmap.

That feature brings a lot of design changes along:

- Introduce `Projects` and associate `Paths` with them. The domain has to know how to track progress for this new kind of data.
- Add background-queue file monitoring capabilities.
- Design new user interface components so the user can manage projects and tracked files' paths.
- Store daily `PathRecords` to keep track of file changes over a long period of time. Without records, no history. Without history, no analysis.
- Make the existing productivity history view aware of `Project` progress.

[5]http://christiantietze.de

For the simple productivity metrics, I store application-specific daily records in `.plist` files. They are easy to set up and get started. But they don't scale very well. Each save results in a full replacement of the file. After 2 years of usages, for example, my record file clocks in at 3 MB. Someone who tracks even more apps and thus increases record history boilerplate will probably have a much bigger file. The hardware will suffer.

In the long run, I'd transition to SQLite or Core Data. The file monitoring module I am about to develop back in 2014 can use its own isolated persistence mechanism. I decide to use Core Data because it's so convenient to use.

Isolating the file monitoring module is one thing. To develop the module with proper internal separation of concerns and to design objects is another. Core Data easily entangles your whole application if you pick the path of convenience. That's usually not a path that scales well once you diverge from the standard way once. In this book, I write about my experience with another path, the path of pushing Core Data to the module or app boundaries and keep the innermost core clean and under control.

Challenges Untangling the Convenient Mess

Back in 2014, I didn't have any experience displaying nested data using `NSOutlineViews`. I didn't have a lot of experience with Core Data. All in all, most of the design decisions and their requirements were new to me.

Teaching myself some AppKit, I fiddled around with providing data to `NSOutlineView` via `NSTreeController`. Cocoa Bindings are super useful, but they're not transparent. When it works, it works; but when it doesn't, it's hard to know why. Getting your hands dirty and acquiring hands-on knowledge is important, though, to get a feeling for the framework. Declarative or book knowledge will only get you so far. I managed to get a few use cases and basic interactions right, like appending projects to the list and adding paths to the projects. So an interface prototype without actual function wasn't too hard to figure out.

Now add Core Data to the equation.

It's super convenient to use Core Data because the framework takes care of *a lot*. If you want to manipulate persistent objects directly from your interface, you can ever wire `NSObjectController` subclasses like the above-mentioned `NSTreeController` to Core Data entities and skip *all of the object creation boilerplate code.* An `NSTreeController` can take care of displaying nested data in an `NSOutlineView`, adding items relative to the user's current selection for example.

If you adhere to the limitations of an `NSTreeController`, that is. It is designed to operate on one type of (Core Data) entity. Unfortunately, users of the Word Counter will edit two kinds of objects, not one: `Paths` may only be nested below `Projects`, which in turn are strict root level objects. I have to enforce these rules myself, and I have to add objects to the tree from code on my own.

To tie Core Data entities (or `NSManagedObject` subclasses) to the user interface is so convenient because it skips all of the layers in between. No view controller, no custom domain objects. **This means, in turn, to couple the user interface to the data representation**. In other words, the user-facing windows are directly dependent on the database. This may be a clever short cut, but this might also be the reason why your app is getting hard to change and hard to test.

The data representation should stay an implementation detail. I may want to switch from Core Data to a SQLite library later on, just like I'm going to migrate my `.plist` files to something else in the future. The app has to be partitioned *at least* into "storing data" and "all the rest" to switch the "storing data" part. To blur all the boundaries is commonly referred to as *Big Ball of Mud* architecture (or rather "non-architecture"). Partitioning of the app into sub-modules improves clarity. And the convenient use of Core Data interferes with this objective.

In short, using Core Data in the most convenient way violates a lot of the principles I learned and loved to adhere to. Principles that helped me keep the Word Counter code base clean and maintainable. Principles I'm going to show you through the course of this book.

The basic architecture I want to advocate is sometimes called "Clean Architecture[6]", sometimes "Hexagonal[7]". There are differences between these two, but I won't be academic about this. This little book will present you my interpretation on these things first and point you into the right direction to find out more afterwards.

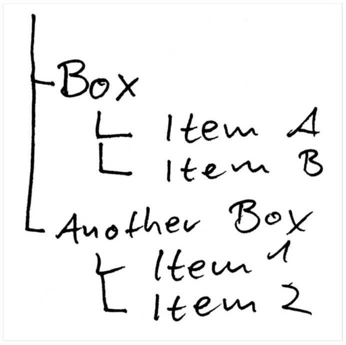
[6]http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html

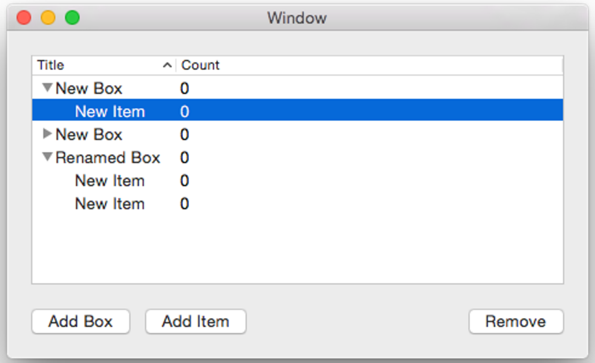
[7]http://alistair.cockburn.us/Hexagonal+architecture

The Example

This book is driven by example. Every pattern used will be put to practice in code.

The example project and this very book you read are the result of me trying to solve a problem in the domain of the Word Counter for Mac. This project is a model, so to speak, to explore solutions.





Title	Count
▼ New Box	0
New Item	0
▶ New Box	0
▼ Renamed Box	0
New Item	0
New Item	0

The tree of entities and a window screenshot

Let's pretend we want to create an application to organize stuff. It manages items. Items reside inside boxes. Users may create boxes and put items inside them. Boxes may only contain items but no other boxes. Items may contain nothing else. It's up to the user to decide what an item is, and she may label each box individually, too.

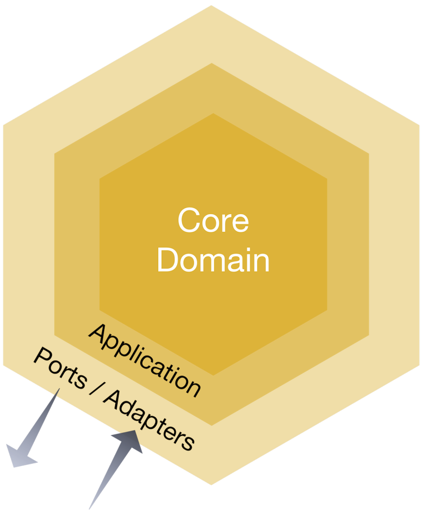
The resulting data can be represented as a very shallow tree structure. That's why we want to display it as a tree view component, using an `NSOutlineView`.

Additionally, we decide to use Core Data for persistence.

The corner stones of the application are set. This is a stage of a lot of app visions when they are about to become a reality: not 100% concrete, not thoroughly designed, but you know the basic course of action. Then you get your hands dirty, explore, and learn. I don't want to advocate a head-over-heels approach here. There's a lot of useful planning you can do in advance that won't prevent innovation later. But this book is not the book to teach you that; this book is aimed to take people from a vision and semi-concrete implementation plan to the next level.

Architecture Overview

Out there, you'll find tons of software architecture approaches and dogmas. I already picked one for this book: a layered clean architecture. It enforces design decisions I found to be very healthy in the long run. With practice, you know where the solution to a problem should be located. And from this context you can derive how to implement something that works. I'll show you how that works throughout the book. For now, I want to introduce the basic framework so we can think in the same terms.



Layered Hexagonal Architecture

For the sake of this book and the rather simple example application, it suffices to think about the following layers, or nested circles:

- The **Domain** at the core of your application, dealing with business rules and actual algorithms.

- The so-called **Application** layer, where interaction with the Domain's services take place.
- **Infrastructure**, hosting persistence mechanisms, and **User Interface**, living in the outermost layer where the app exposes so-called adapters to the rest of the world.

Domain

What's the core of an app?

Is it the user-facing presentational elements? Is it server communication logic? Or is it your money-maker, maybe a unique feature or data crunching algorithm?

I buy into *Domain Driven Design* principles and say: the domain is at the core. It's an abstract thing, really. It's the realm of language; that's where we can talk about "items" that are "organized into boxes." Through setting the stage for the example of this book in the last section, I already created a rough sketch of the domain. This is the *problem space.* A model of the domain of boxes and items in code is part of the *solution space.* These are called "domain objects", and they don't know a thing about the Cocoa framework or Core Data.

Entities in the Domain

`Box` and `Item` are the core parts of this domain. They are **Entities**: objects that have identity over time. Entities are for example opposed to value objects, which are discardable representations. The integer `42` or the string "living-room" are value objects; their equality depends on their value alone. Two Entities with similar values should nevertheless be unequal. If your dog and my dog are both called "Fido", they are still different dogs. Similarly, a user of this app should be able to change the title of two or more existing `Box`es to "living-room" without changing their identity.

A `Box` is a specific kind of Entity. It's an **Aggregate** of entities (including itself): it has label for its own and takes care of a collection of `Item`s. An `Item` will only exist inside a `Box`, so access to it is depending on knowing its `Box`, first. Being an Aggregate is a strong claim about object dependencies and object access. In a naive approach, you'd query the database for boxes and items independently, combine them for display, and that's it. The notion of Aggregates changes your job: you only request boxes and the items are included automatically.

To get `Box` Aggregates, a **Repository** with a collection-like interface is used. The `BoxRepository` specifies the interface to get all boxes, remove existing ones, or add new ones. And when it hydrates a `Box` object from the database, it hydrates the corresponding `Items`, too. Without a Repository, assembling an Aggregate would be very cumbersome. If you know the classic Gang of Four *Design Patterns*, you can think of a Repository in terms of a Factory which takes care of complex object creation.

Services of the Domain

In the domain of the Word Counter, for example, there reside a lot of other objects. Some deal with incoming **Domain Events** like "a word was typed". There are `Recorders` which save these events. `Bookkeepers` hold on to a day's active records. There's a lot of other stuff revolving around the process of counting words.

In this example application, there won't be much going on at the beginning. We will focus on storing and displaying data. That'll be an example for application wire-framing. There'll be one particular **Domain Service** that encapsulates a user intent: to provision new `Boxes` and `Items`. This equals creating new Entities. This Domain Service adds the Entities to the Repository so other objects can obtain them later on.

Services in general are objects without identity that encapsulate behavior. You may argue they shouldn't even have *state*. They execute commands, but they shouldn't answer queries. That's as general a definition I can give without limiting its application too much.

Although I will use the Repository to generate object identifiers through `nextId()`, there is room for improvement. Acting as a storage and vending identifiers are two concerns, we can argue. You could add an `IdentityService` type which only vends new identifiers. This way, the `BoxRepository` will only deal with object retrieval and storage while another service vends identifiers.

Infrastructure

The domain itself doesn't implement database access through repositories. It defines the interfaces or protocols, though. The concrete implementation resides elsewhere: in the **Infrastructure** layer. When the Domain Service reaches "outward" through the Repository interface, it sends data through the Domain's "ports". The concrete Core Data Repository implementation is an "adapter" to one of these ports.

A concrete Repository implementation can be a wrapper for an array, becoming an in-memory storage. It can be a wrapper around a `.plist` file writer, or any other data store mechanism, including Core Data. The Domain doesn't care about the concrete implementation – *that is a mere detail.* The Domain only cares about the fulfillment of the contract; that's why it has 100% control over the protocol definition.

All the Core Data code should be limited to this layer. Exposing the dependency on Core Data elsewhere makes the distinction between Domain and Infrastructure useless. To make the persistence mechanism an implementation detail gives you a lot of flexibility and keeps the rest of the app cleaned from knowledge of this detail. But all the planning and architecture won't help if you blur the boundaries again and sprinkle Core Data here and there because it'd be more convenient.

Application

Your Domain contains all the business logic. It provides interfaces to clients through Aggregates and Services. A layer around this core functionality is called the **Application layer**, because it executes functionality of the Domain.

The Client of the Domain

The Domain contains a Service object to provision new Entities. But someone has to call the Domain Service. The application's user is pressing a button, but how does the button-press translate to using the Domain Services? While we're at it: where does the view controller come from, where is it created and which objects holds a strong reference to it?

This is where the Application layer comes into play. It contains **Application Service** objects which hold on to view controllers and Domain Services, for example. The Application layer is the client of the Domain's functionality.

When the "Add Box" button is pressed, the view controller translates this interaction into a command for the Application layer, which in turn prepares everything to make the Domain provision a new `Box` instance. `Box`es have to be reported to the user interface for display, too. That's what the Application layer will deal with in use case-centered Service objects.

Glue-Code

You may have asked yourself how the Domain will know which Repository implementation to use in the actual program. The Application layer can take care of this when it initializes and uses the Domain's services.

"Dependency Injection[8]" is the fancy term for passing a concrete Repository implementation to an object that requires a certain protocol. It's a trivial concept when you see it in action:

```
func jsonify(serializer: JSONSerializer) -> JSON {
    serializer.add(self.title)
    serializer.add(self.items.map { $0.title })
    return serializer.json()
}
```

Instead of creating the `JSONSerializer` inside the method, the serializer is *injected.* Whenever the function that uses an objects also creates it, you'll have a hard time testing its behavior – and that results in a harder time switching collaborating objects later, too.

Apart from this simple application of passing a collaborator around, Dependency Injection as a paradigm can be used to create object clusters, too. Instead of passing in the dependency with a method call, you pass it as a parameter of the initializer. You probably do things like that in your `AppDelegete` already to wire multiple objects together and pass them to view controllers or navigation controllers. You can use a Dependency Injection library to automate this setup task and configure the dependency network outside your code, but I always found these things to be way too cumbersome. (Then again, I never built huge enterprise software which

[8]http://en.wikipedia.org/wiki/Dependency_injection

may benefit from this kind of flexibility.) This book’s example app isn’t complex enough to warrant more than a global storage object for the main dependencies that are going to be used in the Application Services. The Repository implementations are set up on a **Service Locator** which is usually is a global singleton. I put the Service Locator inside the Infrastructure layer because it mostly consists of Infrastructure parts anyway.

Thus, Infrastructure deals with providing persistence mechanisms to be plugged them into the Domain. Actually calling the methods to set up Domain Services is a matter of the use case objects which I put into the Application layer.

The Story of Developing a Clean App by Example

Part 1: Bootstrapping – Setting Up the Project, Core Data, and Unit Tests

Swift is still pretty new. Since it's so new, it's likely you haven't had a chance to use it seriously. After all, investing in a hip new language is pretty risky: it takes time and might not lead you anywhere.

With Swift, it's a safe bet to stay a Cocoa developer in the future. Apple pushes the language forward, so there's a lot of momentum already. Swift is not in the same situation Java was in during the late 1990's.

Initially, you may not know if errors happen because you can't use Swift properly, or if it's an actual implementation problem. I guess having existing code to port makes it easier to take Swift for a test drive. Without that luxury, you'll have to learn faster.

That's why we're going to start with a *natural* approach to learning Swift and getting an application up and running.

This part of the book is not about designing and crafting the application in a "clean" manner. There's no up-front design. This part is about understanding the new ecosystem of Swift and how it integrates with Xcode and existing Cocoa frameworks. It's an exploratory part: I think you'll be able to follow along with the resistance I encountered and how I thought and coded my way out of dead-ends better in this style.

I dedicate this part to making the switch to Swift and getting everything to run, including:

1. Get Core Data and related tests running
2. Prepare the user interface in its own layer
3. Have tests in place for the app's domain

In the second part, I'll cover the actual application's functionality. We'll be going to explore a lot of architectural patterns in-depth there.

This part is mostly made up of longer notes or journal entries. They aren't meant to tell a coherent story. They are meant to provide a view over my shoulder, include details and background information so you can follow along easily.

Think of this part as a log book to follow the path to adopting Swift. You may want to skim most of this part if you're comfortable with Swift, XCTest, and Core Data already.

Sample Information

Only a small selection of chapters are added to this chapter in this sample. I selected sections which I find most interesting to show you what is covered in the book. Because the sample is missing context, a few things might not make sense the way it is.

Functional Testing Helps Transition to Adding Real Features

I'm pretty confident the view does what it should do. It's all just basic stuff, although it took enough trial & error to reach this point. (I am talking about you, explicit Cocoa Bindings which break the view's behavior!)

I am curious if the app will work, though. This is only natural. Developers want to have a running prototype as early as possible. I'm no exception. Unit tests can help to get feedback that algorithms work. But you won't know if the app executes and performs anything at all until you run it. There are two steps I could take at this point to quench my curiosity.

First, I could use more unit tests to verify that view model updates result in `NSOutlineView` updates. These updates should also be visible when I run the app, but the unit tests won't give visual feedback. Still, I would verify that the controller logic works. Since the data doesn't change in the background but depend on me clicking on widgets, I'm eager to see Cocoa Bindings in action. Then again, I've already witnessed Cocoa Bindings in another project. It felt like magic when the label updated to a new value that the component received after a couple of idle time automatically. Key-Value Observing is doing a lot of the hard work here. I can assure you that the Cocoa Bindings are set up correctly; in fact, the binding tests verify that the setup is working. Seeing the changes in the running app equals a manual integration test – that is testing Apple's internal frameworks to do their job properly. That doesn't make any sense. I would like to see it in action, but it won't provide any useful information.

Second, I could add functional tests to the test harness. And this is what I'll do: in a fashion you usually find with Behavior-Driven Development (as opposed to Test-Driven Development, where you start at the innermost unit level), I'll add a failing test which will need the whole app to work together in order to pass. It's a test which is going to be failing for quite a while. It's an automated integration test that exercises various layers of the app at once.

In the past I wrote failing functional tests for web applications only because they would drive out the REST API or user interaction pretty well. I haven't tried this for iOS or Mac applications, yet. So let's do this.

If you work by yourself, I think it's okay to check in your code with a failing test as long as it's guiding development. Don't check in failing unit tests just because you can't figure out how to fix the problem immediately. Everything goes as long as you don't push the changes to a remote repository. Until that point, you can always alter the commit history.

Having a failing functional test at all may not be okay with your team mates, though. After all, your versioning system should always be in a valid state, ready to build and to pass continuous deployment. Better not check in the failing test itself, then, or work on a local branch exclusively until you rebase your commit history when your feature is ready.

The bad thing about functional tests or integration tests is this: if all you had were functional tests, you'd have to write a *ton* of them. With every condition, with every fork in the path of execution, the amount of functional tests to write grows by a factor of 2. Functional tests grow exponentially. That's bad.

So don't rely too much on them. Unit tests are the way to go.

And make sure you watch J. B. Rainsberger's "Integrated Tests are a Scam"[9] some day soon. It's really worth it.

That being said, let's create a functional test to learn how things work together and guide development.

First Attempt at Capturing the Expected Outcome

I want to go from user interface actions all down to Core Data. That's a first step. This is the test I came up with:

```swift
func testAddFirstBox_CreatesBoxRecord() {
    // Precondition
    XCTAssertEqual(repository.count(), 0, "repo starts empty")

    // When
    viewController.addBox(self)

    // Then
    XCTAssertEqual(repository.count(), 1, "stores box record")
    XCTAssertEqual(allBoxes().first?.title, "New Box")
}
```

There's room for improvement: I expect that there's a record with a given `BoxId` afterwards, and that the view model contains a `BoxNode` with the same identifier. This is how the view, domain, and Core Data stay in sync. But the naive attempt at querying the `NSTreeController` is hideous:

```swift
viewController.itemsController.arrangedObjects.childNodes!!.first.boxId
```

I rather defer such tests until later to avoid these train wreck-calls and stick with the basic test from above that at least indicates the `count()` did change, even though I don't know if the correct entity was inserted at this point.

[9] https://vimeo.com/80533536

Making Optionals-based Tests Useful

On a side note, I think Swift's optionals are making some tests weird. With modern Swift, optional chaining and `XCTAssertEqual` play together nicely:

```
XCTAssertEqual(allBoxes().first?.title, "New Box")
```

Sometimes, you need to unwrap an optional, though. Force-unwrapping `nil` results in a runtime error which we want to avoid during tests because it interrupts execution of the whole test suite. A failure is preferable.

```
if let box = allBoxes().first {
    XCTAssertEqual(box.title, "New Box")
} else {
    XTCFail("expected 1 or more boxes")
}
```

Imagine `allBoxes()` returned an optional; then the complexity of the test would increase a lot:

```
if let boxes = allBoxes() {
    if let box: ManagedBox = allBoxes().first {
        XCTAssertEqual(box.title, "New Box")
    } else {
        XCTFail("no boxes found")
    }
} else {
    XCTFail("boxes request invalid")
}
```

When you work with collections, instead of `nil`, return an empty array. This makes the result much more predictable for the client. Here's the updated version:

Don't allow optionals if there's no need to

```swift
func allBoxes() -> [ManagedBox] {
    let request = NSFetchRequest(entityName: ManagedBox.entityName())
    let results: [AnyObject]

    do {
        try results = context.fetch(request)
    } catch {
        XCTFail("fetching all boxes failed")
        return []
    }

    guard let boxes = results as? [ManagedBox] else {
        return []
    }

    return boxes
}
```

If optional chaining doesn't work for you for some reason, make it a two-step assertion instead:

```swift
let box: ManagedBox = allBoxes().first
XCTAssertNotNil(box)
if let box = box {
    XCTAssertEqual(box.title, "New Box")
}
```

If the value is `nil`, the first assertion will fail and the test case will be marked as a failing test. You can come back to it and fix it. No breaking runtime errors required!

Wire-Framing the Path to "Green"

Now this integration test fails, of course. In broad strokes, this is what's left to do to connect the dots and make it pass:

- I have to make a repository available to the view. I'll do this via **Application Services**.
- I have to add an actual Application Service layer. Remember, this is the client of the domain.
- The Application Service will issue saving `Boxes` and `Items` without knowing about Core Data.
- I need a service provider of sorts. Something somewhere has to tell the rest of the application that `CoreDataBoxRepository` (from Infrastructure) is the default implementation of the `BoxRepository` protocol (from the Domain). The process of setting this up takes place in the application delegate, but there's a global `ServiceLocator` singleton missing to do the actual look-up.
- I may need to replace the `ServiceLocator`'s objects with test doubles.

The `ServiceLocator` can look like this:

ServiceLocator singleton to select default implementations

```
open class ServiceLocator {
    open static let sharedInstance = ServiceLocator()

    // MARK: Configuration

    fileprivate var managedObjectContext: NSManagedObjectContext?

    public func setManagedObjectContext(_ managedObjectContext: NSManagedObj\
ectContext) {
        precondition(self.managedObjectContext == nil,
            "managedObjectContext can be set up only once")

        self.managedObjectContext = managedObjectContext
    }

    // MARK: Dependencies

    public class func boxRepository() -> BoxRepository {
        return sharedInstance.boxRepository()
    }

    // Override this during tests:
    open func boxRepository() -> BoxRepository {
        guard let managedObjectContext = self.managedObjectContext
            else { preconditionFailure("managedObjectContext must be set up"\
) }

        return CoreDataBoxRepository(managedObjectContext: managedObjectCont\
ext)
    }
}
```

It’s not very sophisticated, but it is enough to decouple the layers. An Application Service could now perform insertions like this:

```
public func provisionBox() -> BoxId {
    let repository = ServiceLocator.boxRepository()
    let boxId = repository.nextId()
    let box = Box(boxId: boxId, title: "A Default Title")

    repository.addBox(box)

    return boxId
}
```

This is not a command-only method because it returns a value. Instead of returning the new ID so the view can add it to the node, the service will be responsible for actually adding the node to the view. That’d be the first major refactoring.[10]

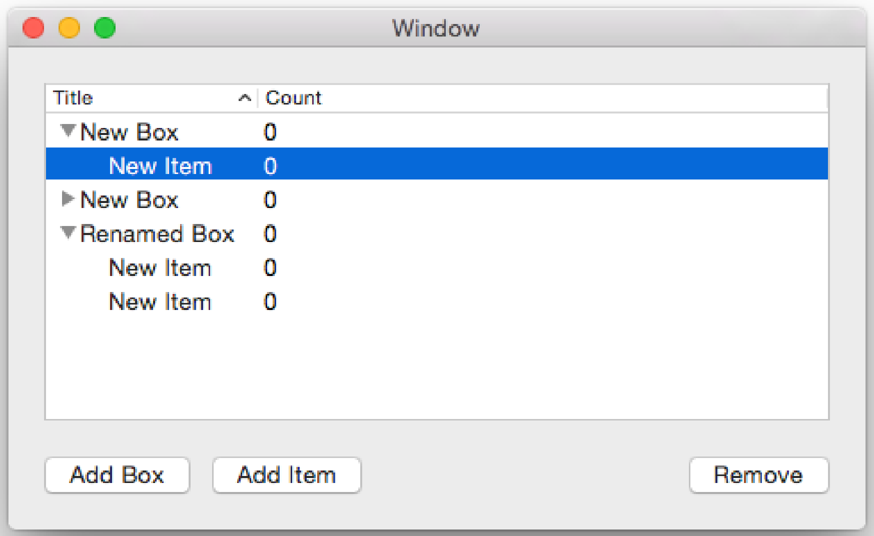
It’s about time to leave the early set-up phase and enter Part II, where I’m going to deal with all these details and add the missing functionality.

[10]See commit cf86167

Part 2: Sending Messages Inside of the App

I tried to prepare all of the app's components so far in isolation and not worry much about the integration. In other words, running the app will show that things don't seem to work. The basic structure is in place, but there's virtually no *action*. That's changing in this part.

The app window is going to look like this:

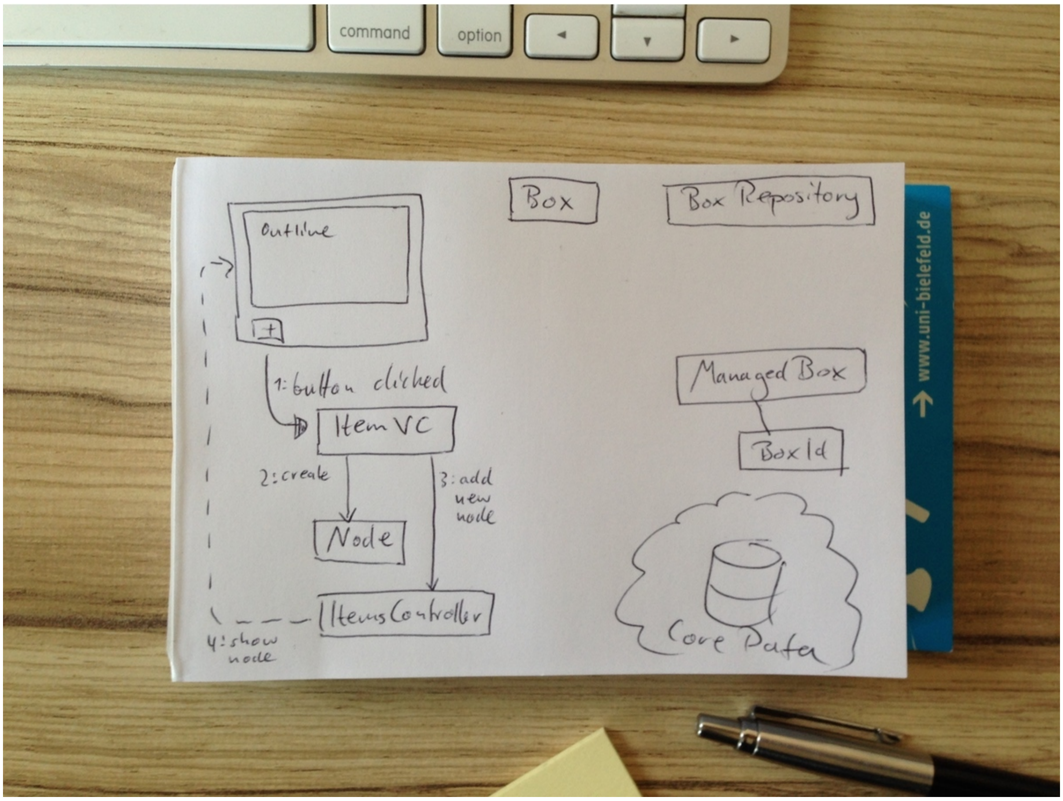


Title	Count
▼New Box	0
New Item	0
▶New Box	0
▼Renamed Box	0
New Item	0
New Item	0

Final App Window

At the moment, the state of the app is the following:

- The user interface is operational but doesn't persist data. The `ItemViewController` reacts to button presses and creates node objects.
- The Core Data part of infrastructure seems to work fine according to the tests but doesn't receive any commands, yet.
- The domain is particularly boring. It has no behavior at all. Until now, the whole demo application is centered around getting user input to the data store. This will not change until the next part. It's nonsensical to work this way of course when the domain should be the very key component of the application.
- There's no event handling at all, as in "User adds a Box to the list". There's no layer between user interface and domain.



Components

This part will focus on the integration and various methods of passing messages between components of the app. When I figured out the *structure* of the app in the last sections, now I worry more about designing the *processing* of information and performang actions. There are a few attractive options to consider:

- Cocoa's classic delegate pattern to obtain objects ("data source") and handle interactions ("delegate")
- Ports & Adapters-style command-only interfaces, adhering to CQRS[11]
- Leveraging Domain Events (since there's no domain worth speaking of, we'll defer exploring that option to part 3)

We'll look at all of these in detail.

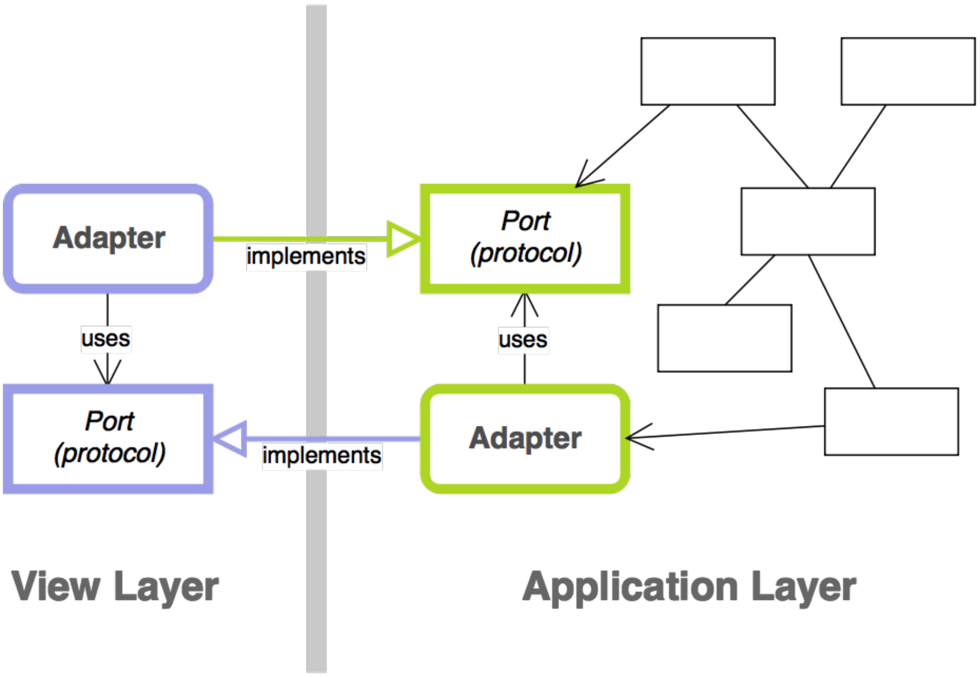
Sample Information

Only a small selection of chapters are added to this chapter in this sample. I selected sections which I find most interesting to show you what is covered in the book. Because the sample is missing context, a few things might not make sense the way it is.

[11]According to "Command–Query Responsibility Segregation" you separate changes (commands) from obtaining data (queries). A method should return a value or cause a change in the system, never both.

Ports and Adapters

Recall that the basic idea of this architectural style is this: separate queries from commands and isolate layers through ports and adapters. *Ports* are interfaces (protocols) declared in one layer, while *Adapters* are classes from other layers which satisfy a Port's specification:



Ports and Adapters

Instead of one adapter knowing about its collaborator on the other side of the boundary, each adapter only knows a protocol from its module. That marks a dependency. The actual implementation of that protocol is *injected*. This move is sometimes called “Dependency Inversion,” because you don't go the naive route and depend on a component from another module but instead depend on a protocol from your own module that others satisfy. The “inversion,” then, is the inversion of arrow directions.

This is nothing new to the code, actually. For example, `BoxRepository` is a port of the domain to which `CoreDataBoxRepository` from infrastructure is an adapter. Similarly, the `HandlesItemListEvents` protocol from the user interface layer is implemented by an application service.

I want to refactor `HandlesItemListEvents` to pay more attention to command–query separation.

Concerns With the Naive Approach

Have a look at the event handling protocol again:

```
provisionNewBoxId() -> BoxId
```

In fact, this method is kind of a mix between typical data source and delegate method. It's intent is a command, suitable for user interaction events. But it's also meant to return an object like a factory or data source does.

It should be rephrased as such:

```
provisionBox()
newBoxId() -> BoxId
```

The two processes can't be split up in the **Application Service**, though. With the current expectations in place, `newBoxId()` would have to return the last ID the service had provisioned. This way, everything will fall apart too easily once `provisionBoxId()` is called twice. Just think about concurrency. The contract to call both methods in succession only can't be enforced, so better not rely on it.

Another alternative is to model it as such:

```
newBoxId() -> BoxId
provisionBox(_: BoxId)
```

The service will be an adapter to both domain and infrastructure this way. It'd work, but it's not what I aim for.

To model the expectation of "provision first, obtain ID second" more explicitly, one could introduce a callback to write the ID to, like so:

```
provisionBox(andReportBoxId: (BoxId) -> Void)
```

I don't like how that reads, though. And since there's just a single window present, we can formalize this as another protocol from the point of view of `BoxAndItemService`.

My very first take:

Introduce consumer protocol as output port

```swift
protocol ConsumesBoxId: class {
    func consume(boxId: BoxId)
}

class BoxAndItemService: HandlesItemListEvents {
    // Output port:
    var boxIdConsumer: ConsumesBoxId?

    func provisionBox() {
        let repository = ServiceLocator.boxRepository()
        let boxId = repository.nextId()
        storeNewBox(withId: boxId, into: repository)
        reportToConsumer(boxId: boxId)
    }

    fileprivate func storeNewBox(withId boxId: BoxId,
        into repository: BoxRepository) {

        let box = Box(boxId: boxId, title: "New Box")
        repository.addBox(box)
    }

    fileprivate func reportToConsumer(boxId: BoxId) {
        // Thanks to optional chaining, a non-existing boxIdConsumer
        // will simply do nothing here.
        boxIdConsumer?.consume(boxId: boxId)
    }
}
```

Instead of *querying* `BoxAndItemService` for an ID, the view controller can now *command* it to provision a new `Box`. The service won't do anything else. Clicking the button will add a `Box` as it says on the label, but it won't change the view. The design of these components is flexible and the service doesn't know the consumer. It is not tied to a view component. You could say that it's just a coincidence the `ItemViewController` in return receives the *command* to `consume()` a `BoxId`. This, in turn, will trigger adding a new node with the appropriate `BoxId` to the outline view.

Using Domain Services and Domain Events

What `reportToConsumer(boxId:)` does equals the intent of the well-known *observer pattern*. In Cocoa, we usually send notifications. This is more like calling a delegate because it's a 1:1 relationship instead of the many-to-one observer pattern relationship:

```
func reportToConsumer(box: Box) {
    boxConsumer?.consume(box)
}
```

But I notice the **Application Service** `BoxAndItemService` is now doing these things:

- it sets up the aggregate `Box`
- it adds the aggregate instance to its repository
- it notifies interested parties (limited to 1) of additions

Essentially, that's the job of a **Domain Service**.

The domain has a few data containers, but no means to create **Aggregate**s or manipulate data. The application layer, being client to the domain, shouldn't replace domain logic. Posting "Box was created" events is the domain's responsibility.

Using `NotificationCenter` as a domain event publisher, `BoxAndItemService` loses part of its concerns in favor of a Domain Service, `ProvisioningService`:

Refactoring business logic into a dedicated Domain Service

```
open class ProvisioningService {
    let repository: BoxRepository

    var eventPublisher: NotificationCenter {
        return DomainEventPublisher.defaultCenter()
    }

    public init(repository: BoxRepository) {
        self.repository = repository
    }

    open func provisionBox() {
        let boxId = repository.nextId()
        let box = Box(boxId: boxId, title: "New Box")

        repository.addBox(box)

        eventPublisher.post(
            name: Events.boxProvisioned,
            object: self,
            userInfo: ["boxId" : boxId.identifier])
    }

    open func provisionItem(inBox box: Box) {
        let itemId = repository.nextItemId()
        let item = Item(itemId: itemId, title: "New Item")

        box.addItem(item)

        let userInfo = [
            "boxId" : box.boxId,
            "itemId" : itemId
        ]
        eventPublisher.post(
            name: Events.boxItemProvisioned,
            object: self,
            userInfo: userInfo)
    }

    // (Mis)using enum as a namespace for notifications:
    enum Events {
        static let boxProvisioned =
            Notification.Name(rawValue: "Box Provisioned")
        static let boxItemProvisioned =
            Notification.Name(rawValue: "Box Item Provisioned")
    }
}
```

I'm not all that keen about the way `NotificationCenters` work. Dealing with the `userInfo` parameter is error-prone; both during consumption and creation you can have a typo in a key and end up with a runtime error or unexpected behavior.

I think `provisionItem` doesn't read too well, but it's not the worst method in human history, either. But getting the data out is getting bad:

```
let boxInfo = notification.userInfo?["boxId"] as! NSNumber
let boxId = BoxId(fromNumber: boxInfo)
```

It's hard to read and complicated when compared to, say:

```
let boxId = boxCreatedEvent.boxId
```

To introduce another layer of abstraction is a good idea, especially since Swift's structs make it really easy to create **Domain Event**s. It will improve the code, although you'll have to weigh the additional cost of developing and testing this very layer of abstraction. For the sake of this sample application, I prefer not to over-complicate things even more and stick to plain old notifications for now.

To replace the command–query-mixing methods from before, there needs to be an event handler which subscribes to Domain Events in the application layer and displays the additions in the view.

Before I expand the Domain with these event types, though, we'll have a look at another refactoring in the Application layer.

Part 3: Putting a Domain in Place

For the most part of this exercise, the Domain consisted of two **Entities**: `Box` and `Item`. It wasn't a domain model worth talking about. There were data containers without any behavior at all. There were no business rules except managing `Items` in `Boxes`.

This changed a bit when I realized I had mixed **Domain Service** and **Application Service** into a single object. The resulting `ProvisioningService` in the domain creates Entities, adds them to the repository, and notifies interested parties of the event.

Notifications are useful for auto-updating the view as I mentioned in the last part already. They are useful for populating an event store, too: persist the events themselves instead of Entity snapshots to replay changes and thus synchronize events across multiple clients, for example. This is called Event Sourcing[12] and replaces traditional database models to persist Entity states. Digging into this goes way beyond the scope of this book, but I'm eager to try it in the future (hint, hint).

Sample Information

Only a small selection of chapters are added to this chapter in this sample. I selected sections which I find most interesting to show you what is covered in the book. Because the sample is missing context, a few things might not make sense the way it is.

Introducing Events in Place of Notifications

In the last part, I ended up using `NotificationCenter` to send events. Notifications are easy to use and Foundation provides objects that are well-known. I don't like how

[12] http://msdn.microsoft.com/en-us/library/dn589792.aspx

working with a notification's `userInfo` dictionary gets in the way in Swift, though. Objective-C was very lenient when you used dictionaries. That introduced a new source of bugs, but if you enjoy dynamic typing, it worked very well. Swift seems to favor new paradigms that enforce strong typing.

Swift Is Sticking Your Head Right at the Problem

In Objective-C, I'd send and access the "Box was created" event info like this:

```
// Sending
NSDictionary *userInfo = @[@"boxId": @(boxId.identifier)];
[notificationCenter postNotificationName:kBoxProvisioned,
                                  object:self,
                                userInfo:userInfo];

// Receiving
int64_t identifier = notification.userInfo["boxId"].longLongValue;
[BoxId boxIdWithIdentifier:identifier];
```

Swift 3 allows sending notifications with with number value types directly. We don't have to wrap them in `NSNumber` anymore. Still, the force-unwrapping and the forced cast make me nervous in Swift because they point out a brittle piece of code:

```
// Sending
let userInfo = ["boxId" : boxId.identifier]
notificationCenter.post(name: kBoxProvisioned,
    object: self, userInfo: userInfo)

// Receiving
let boxInfo = notification.userInfo!["boxId"] as! NSNumber
let identifier = boxInfo.int64Value
let boxId = BoxId(identifier: identifier)
```

I settled for sending IDs only because putting ID and title makes things complicated for the "Item was created" event. There, I'd have to use nested dictionaries. A JSON representation would look like this:

JSON representation of the event data

```
{
    box: {
        id: ...
    }
    item: {
        id: ...
        title: "the title"
    }
}
```

Accessing nested dictionaries in Swift is even worse, though, so I settled with supplying two IDs only. On the downside, every client now has to fetch data from the repository to do anything with the event. That's nuts.

The relative pain I experience with Swift here highlights the problems Objective-C simply assumed we'd take care of: there could be no `userInfo` at all, there could be no value for a given key, and there could be a different kind of value than you expect.

It's always a bad idea to simply assume that the event publisher provided valid data in dictionaries. Force-unwrapping and force-casting will accidentally break sooner or later. What if you change dictionary keys in the sending code but forgot to update all client sites? Defining constants remedies the problem a bit. But the structure of a dictionary is always opaque, and if you change it, you have to change multiple places in your code. It's a good code heuristic to look for changes that propagate through your code base. If you have to touch more than 1 place to perform a change, that's an indicator of worse than optimal encapsulation: these co-variant parts in your app depend on one another but don't show their dependency explicitly.

So you have to perform sanity checks to catch invalid events anyway, in Objective-C just as much as in Swift.

Using real event objects will work wonders. Serializing them into dictionaries and de-serializing `userInfo` into events will encapsulate the sanity checks and provide usable interfaces tailored to each event's use. There's only one place you need to worry about if you want to change the nature of an event.

Event Value Types

An event should be a value type, and thus a struct. It assembles a `userInfo` dictionary. For `NotificationCenter` convenience, it also assembles a `Notification` object:

Domain Event serializing itself into userInfo dictionary

```swift
// Provide a typealias for brevity and readability
public typealias UserInfo = [AnyHashable : Any]

public struct BoxProvisionedEvent: DomainEvent {

    public static let eventName = Notification.Name(
        rawValue: "Box Provisioned Event")

    public let boxId: BoxId
    public let title: String

    public init(boxId: BoxId, title: String) {
        self.boxId = boxId
        self.title = title
    }

    public init(userInfo: UserInfo) {
        let boxIdentfier = userInfo["id"] as! IntegerId
        let title = userInfo["title"] as! String
        self.init(boxId: BoxId(boxIdentfier), title: title)
    }

    public func userInfo() -> UserInfo {
        return [
            "id" : boxId.identifier,
            "title" : title
        ]
    }
}
```

The underlying protocol is really simple:

```
public protocol DomainEvent {
    static var eventName: Notification.Name { get }

    init(userInfo: UserInfo)
    func userInfo() -> UserInfo
}
```

When publishing an event, these properties can be used to convert to a `Notification`. I used to use a free `notification(_:)` function for that:

```
func notification<T: DomainEvent>(event: T) -> Notification {
    return Notification(name: T.eventName, object: nil,
        userInfo: event.userInfo())
}
```

Now with protocol extensions, the conversion can be coupled more closely to the `DomainEvent` protocol, getting rid of the free function:

```
extension DomainEvent {
    public func notification() -> Notification {
        return Notification(
            name: type(of: self).eventName,
            object: nil,
            userInfo: self.userInfo())
    }

    func post(notificationCenter: NotificationCenter) {
        notificationCenter.post(self.notification())
    }
}
```

See the convenience method `post(notificationCenter:)`? That can come in handy when testing the actual sending of an event if you override it in the tests.

Now `BoxProvisionedEvent` wraps the `Notification` in something more meaningful to the rest of the app. It also provides convenient accessors to its data, the ID and title of the newly created box. That's good for slimming-down the subscriber: no need to query the repository for additional data.

There's a `DomainEventPublisher` which takes care of the actual event dispatch. We'll have a look at that in a moment. With all these changes in place, the `DisplayBoxesAndItems` Application Service now does no more than this:

```swift
class DisplayBoxesAndItems {
    var publisher: DomainEventPublisher! {
        return DomainEventPublisher.sharedInstance
    }

    // ...

    func subscribe() {
        let mainQueue = OperationQueue.mainQueue()

        boxProvisioningObserver = publisher.subscribe(
            BoxProvisionedEvent.self, queue: mainQueue) {
                [weak self] (event: BoxProvisionedEvent!) in

                let boxData = BoxData(boxId: event.boxId, title: event.title)
                self?.consumeBox(boxData)
        }

        // ...
    }

    func consumeBox(boxData: BoxData) {
        consumer?.consume(boxData)
    }

    // ...
}
```

The `subscribe` method is interesting. Thanks to Swift generics[13], I can specify an

[13]http://swiftyeti.com/generics/

event type using `TheClassName.self` (the equivalent to `[TheClassName class]` in Objective-C) and pipe it through to the specified block to easily access the values.

The conversion of `Notification` to the appropriate domain event takes place in the `DomainEventPublisher`:

```
 1 func subscribe<T: DomainEvent>(
 2     _ eventKind: T.Type,
 3     queue: OperationQueue,
 4     usingBlock block: (T) -> Void)
 5     -> DomainEventSubscription {
 6
 7     let eventName: String = T.eventName
 8     let observer = notificationCenter.addObserver(forName: eventName,
 9         object: nil, queue: queue) {
10         notification in
11
12         let userInfo = notification.userInfo!
13         let event: T = T(userInfo: userInfo)
14         block(event)
15     }
16
17     return DomainEventSubscription(observer: observer, eventPublisher: self)
18 }
```

It takes some getting used to Swift to read this well. I'll walk you through it.

Let's stick to the client code from above and see what subscribing to `BoxProvisionedEvents` does:

- The type of the `eventKind` argument should the type (not instance!) of a descendant of `DomainEvent`. That's what `BoxProvisionedEvent.self` is. You don't pass in an actual event, but it's class (or "type"). Interestingly, this value is of no use but to set `T`, the generics type placeholder.
- The `block` (line 3) yields an event object of type `T` (which becomes an instance of `BoxProvisionedEvent`, for example)

- The `eventName` (line 4) will be, in this example case, `Box Provisioned Event`. `DomainEvents` have a property called `eventName` to return a string which becomes the notification name.
- The actual observer is a wrapper around the `block` specified by the client. The wrapper creates an event of type `T`. All `DomainEvents` must provide the deserializing initializer `init(userInfo: [Hashable : Any])`, and so does `T`.

When a `BoxProvisioned` event is published, it is transformed into a `Notification`. The notification is posted as usual. The wrapper around the client's subscribing block receives the notification, de-serializes a `BoxProvisioned` event again, and provides this to the client.

`DomainEventSubscription` is a wrapper around the observer instances `NotificationCenter` produces. This wrapper unsubscribes upon deinit automatically, so all you have to do is store it in an attribute which gets nilled-out at some point.

It took some trial and error to get there, but it works pretty well.[14]

[14]See the latest commit, or the initial commits d0a8c7b, 9a8f41d, a2b0f4c, and 74cf804

Appendix

The appendix is mostly a collection of blog post-sized writings about problems I discovered. Apart from the upcoming list of suggested reading, the articles are in no particular order.

Sample Information

Only a small selection of chapters are added to this chapter in this sample. I selected sections which I find most interesting to show you what is covered in the book. Because the sample is missing context, a few things might not make sense the way it is.

Further Reading

Interesting Links

You should really check out the following:

SourceView sample code (Objective-C)

https://developer.apple.com/library/mac/samplecode/SourceView/Listings/BaseNode_m.html#//apple_ref/doc/uid/DTS10004441-BaseNode_m-DontLinkElementID_6

Example application by Apple where `NSOutlineView` is used. Good for getting started, although the example app doesn't run properly anymore – I don't see any images in the source view.

Clean Architecture

http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html

http://blog.8thlight.com/uncle-bob/2011/11/22/Clean-Architecture.html

Uncle Bob's explanation of the architectural concepts I tried to adhere to. Goes beyond Hex

Uncle Bob: Architecture – The Lost Years (Video)

https://www.youtube.com/watch?v=WpkDN78P884

This talk introduced me to Hexagonal and Clean Architecture.

Jim Gay: Clean (Ruby) Code

Video: East-Oriented Code[15] at RubyConf 2014

Book: "Clean Ruby"[16] – full of good examples which you can apply to Swift, too, and even more so to Objective-C and its meta-programming capabilities.

[15] http://confreaks.com/videos/4825-RubyConf2014-eastward-ho-a-clear-path-through-ruby-with-oo

[16] http://clean-ruby.com

Avdi Grimm: Objects on Rails

http://objectsonrails.com

A free e-book which taught me how to decouple my application from Ruby on Rails. This one got me hooked on thinking past using the framework properly.

Other links of interest on the topic of software architecture:

- "250 Days Shipping With Swift and VIPER"[17] (Video)
- "Integrated Tests are a Scam"[18] (Video)
- "Advanced iOS Application Architecture and Patterns", Session 229 at WWDC 2014[19] (Video)
- "Building Better Apps with Value Types in Swift"[20] (Video) at WWDC 2015
- What service objects are not[21], with Ruby examples

[17]https://realm.io/news/altconf-brice-pollock-250-days-shipping-with-swift-and-viper/

[18]https://vimeo.com/80533536

[19]https://developer.apple.com/videos/wwdc/2014/

[20]https://developer.apple.com/videos/wwdc/2015/?id=414

[21]https://medium.com/@KamilLelonek/what-service-objects-are-not-7abef8aa2f99

Interesting Books

I recommend reading the following books, ordered by subjective significance to the topic.

Vaughn Vernon (2013): *Implementing domain driven design*, Upper Saddle River, NJ: Addison-Wesley.
— Lots of practical examples and explanation for the patterns Evans laid out.

Eric Evans (2006): *Domain-Driven Design. Tackling complexity in the heart of software*, Upper Saddle River, NJ: Addison-Wesley.
— Full of good examples and refactorings itself but a little light on actually solving implementation problems.

Michael C. Feathers (2011): *Working effectively with legacy code*, Upper Saddle River, NJ: Prentice Hall Professional Technical Reference.
— Helps to learn decoupling code incrementally and how to test hard-to-test parts. That's where I learned to provide means to reset singletons, and that it's better to wrap `NotificationCenter` *in order to replace the wrapper in tests than not test notifications at all.*

Steve Freeman and Nat Pryce (2010): *Growing object-oriented software, guided by tests*, Boston: Pearson Education.
— This book has taught me so much about Test-Driven Development! I figured out when to use functional tests, why it's beneficial to start with a failing test on the system level to guide development, and how to create multi-layered applications with test fakes all thanks to this book.

Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard (1990): *Object-Oriented Software Engineering. A Use Case Driven Approach*, Wokingham: Addison-Wesley.
— *Actually a recommendation by Uncle Bob in _Architecture: The Lost Years*. It taught me to think about application services and use case objects. The book's focus is on architecture, not on code. Useful practices. You may get this one used for a few dollars. I found it in my local university's library._

Scott Millett (2014): *Practicing Domain-Driven Design. Practical advice for teams implementing the development philosophy of Domain-Driven Design. With code examples in C# .NET*, Scott Millett.

Robert C. Martin (2009): *Clean Code. A Handbook of Agile Software Craftsmanship*, Upper Saddle River: Prentice Hall.

David West (2004): *Object thinking*, Redmond, Wash.: Microsoft Press.

Andy Oram and Greg Wilson (Eds.) (2007): *Beautiful Code*, Beijing: O'Reilly.

Sandi Metz (2013): *Practical object-oriented design in Ruby: an agile primer*, Upper Saddle River, NJ: Addison-Wesley.
— *A really good read. Sadly, it's about Ruby, not about Swift or Objective-C, but you may want to peek into it anyway.*