

objc ↑↓ App Architecture

iOS Application Patterns in Swift

By Chris Eidhof, Matt Gallagher, and Florian Kugler

Version 0.1 (January 2018)

© 2018 Kugler und Eidhof GbR
All Rights Reserved

For more books and articles visit us at <http://objc.io>
Email: mail@objc.io
Twitter: [@objcio](https://twitter.com/objcio)

About this book 4

1 Introduction 7

Application Architecture 7

Separating into Layers 8

Applications are a Feedback Loop 9

Architectural Technologies 11

Application Design Patterns 11

2 Overview of Application Design Patterns 14

Model-View-Controller 15

Online-Only Model-View-Controller 18

Model-View-ViewModel+Coordinator 21

The Elm Architecture 25

Model-View-Controller + ViewState (MVC+VS) 29

ModelAdapter-ViewBinder 31

Patterns Not Covered 35

3 Model-View-Controller 39

Exploring The Implementation 40

Testing 53

Discussion 58

Improvements 59

Conclusion 72

4 Model-View-ViewModel + Coordinator 74

Exploring the implementation 76

Testing 92

Discussion 92

Lessons to be Learned 99

About this book

Warning: *This is a pre-release, the final contents will change. The text isn't copy-edited yet, we apologize for spelling and grammatical errors.*

The book is structured around six implementations of the same application, each implementation using a different application design pattern.

Top down

Application design patterns are the highest level of application architecture, describing an overall shape and pattern of data flow for the application. Application design patterns can be a simple topic to read – filled with basic block diagrams and catchy pattern names. You have probably seen acronyms like MVC and MVVM. You might have read about less common terms like “Unidirectional Data Flow”.

But application design patterns do not describe application implementations.

Most application design patterns avoid specifying technologies, libraries, low-level patterns or techniques. Does an arrow on a diagram mean a function call? An observation? A callback function? A delegate pattern? In most cases, these answers are not specified by the pattern. Yet we have implemented whole programs according to application design patterns and these programs are very different at a source code level. If application design patterns are so vague and ill-defined, why are these implementations so different?

The answer is that subtle philosophical ideas accompany each pattern. The techniques chosen for each pattern to solve problems of control flow, data dependencies, ownership, state, compile-time information and interface structure are influenced by these philosophies. Further, there is a culture around programming and cultures develop conventions and our programs reflect these conventions.

Accordingly, the six implementations in this book will be a look at different programming conventions, cultures, needs and philosophies at the same time as they are a look at the patterns, themselves.

Bottom up

The problems faced in application development are solved through the techniques used for control flow, data dependencies, ownership, state, compile-time information and interface structure – the architectural techniques – not by application design patterns themselves.

Though this book is structured around application design patterns, it is not about choosing to adopt patterns in their entirety as much as it is an effort to learn from each pattern and how to best combine ideas and techniques for solving problems in our own programs.

Reducers, reactive programming, interface decoupling, state enums, multi-model abstractions – these ideas may be associated with specific patterns but we will look at how these techniques can apply across different patterns as needed to solve problems.

As this book will show, application architecture is a topic with multiple solutions to every problem. Properly implemented, all solutions will give the same result to the end-user. This means that ultimately, application architecture is a set of choices to satisfy ourselves as programmers: what problems do we want solved implicitly, what problems do we want to consider on a case-by-case basis, where do we need freedom, where do we need consistency, where do we want abstraction, where do we want simplicity?

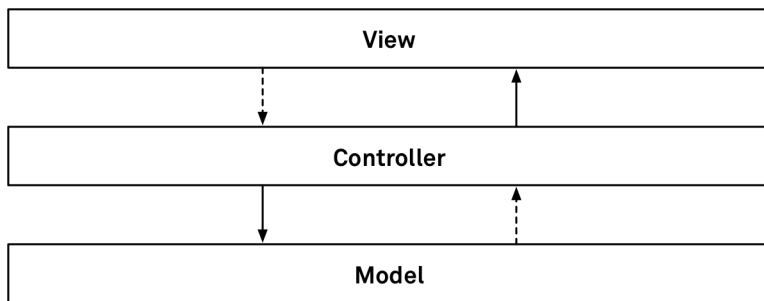
Introduction

1

Application Architecture

Application architecture is a branch of software design concerned with the structure of an application: how applications are divided into different interfaces and conceptual layers; and the control flow and data flow paths taken by different operations between and within these different components.

We often use simple block diagrams to explain application architectures at the highest level. For example, Apple's Model-View-Controller pattern describes three layers: the model, the view, and the controller layer.



The blocks in this diagram show the different named layers of the pattern – the majority of the code you write in a Model-View-Controller project fits into one of these layers. The arrows show how the layers are connected.

However, a simple block diagram explains very little about how the pattern is expected to operate in practice. An application architecture includes many expectations about how components should be constructed, how events flow through the layers, whether components should have compile-time or runtime references to each other, how data in different components should be read or mutated and what paths changes should take through the application structure.

Separating into Layers

Model and View

At the highest level, an application architecture is a basic set of categories into which different application components are separated. In this book, we call these different categories *layers*: a collection of interfaces and other code that conform to some basic rules and responsibilities.

The most common of these categories are the model layer and the view layer.

The *model layer* is an abstract description of the application's contents, without any dependence upon the application framework, offering full control to the programmer. The model layer typically contains model objects (examples in the Recordings app include folders and recordings) and often contains coordinating objects as well (for example, objects that persist data on disk, such as the Store in our example application).

The *view layer* is the application framework dependent part that makes the model layer visible and user interactable, turning the model layer into an application. When writing iOS applications, the view layer almost always uses UIKit directly. However, as we will see, some architectures have a different view layer that wraps around UIKit. For some custom applications, notably games, the view layer might not be UIKit or AppKit: it could be SceneKit or a wrapper around OpenGL.

Sometimes, model or view instances are represented by structs or enums rather than objects, but for the abstract discussion, we'll ignore that distinction. When we write *model objects*, they might manifest as objects, structs or enums, and likewise for *view objects*.

View objects typically form a single *view hierarchy* where all objects are connected in a tree structure with the screen at the trunk, windows within the screen and increasingly smaller views at the branches and leaves. Likewise, view controllers also typically form a *view controller hierarchy*. Model objects don't necessarily form a hierarchy – there could be unrelated models in the program with no connection between them.

When we write *view*, we usually mean a single view object, such as a button or a label. When we write *model*, we usually mean a single model object, such as a Recording or Folder instance. In most literature on the subject, “the model” means different things

depending on the context. It could mean “the model layer”, “the concrete objects that are in the model layer”, or a discrete “document” within the model layer. At the cost of verbosity, we try to be explicit about the different meanings.

Why Are the Categories of Model and View Considered So Fundamental?

It is certainly possible to write an application where there’s no separation between model and view layer. Simple modal dialogs are an example of user interface elements that commonly lack any separate model data, often choosing to simply read the state directly from user-interface elements when the “OK” button is used. In general though, without a separated model layer, it’s extremely difficult to ensure that actions in the program occur according to any coherent rules.

More than anything else, we define a model layer so that we have a single source of truth in our program that is clean and well behaved and not governed by the implementation details of the application framework.

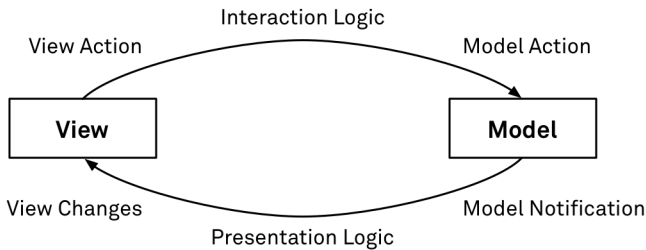
An *application framework* provides infrastructure upon which we can build applications. In this book we use Cocoa – more specifically, UIKit, AppKit or WatchKit, depending on the target platform – as the application framework.

If the model layer is kept separate from the application framework, we can use it completely outside the bounds of the application. We can easily run it in a separate testing harness or we can write a new view layer using a different application framework and use our model layer in the Android, macOS or Windows version of the application.

Applications are a Feedback Loop

The view layer and the model layer need to communicate. There are, therefore, connections between the two. Assuming the view layer and model layer are clearly separated and not inextricably linked, communication between the two will require some form of translation.

These points lead to an abstract description of actions in an application with the following structure:



The result is a feedback loop – not surprising since a user-interface with both display and input functionality is fundamentally a feedback device. The challenge for each application design pattern is how to handle the communication, dependencies and transformations inherent in the arcs shown in this diagram.

Different parts of the path between the model layer and view layer have different names. The name *view action* refers to the code path triggered by a view in response to a user initiated event, such as a tapping a button, or selecting a row in a table view. When a view action is sent through to the model layer, it may be converted into a *model action* (an instruction to a model object to perform an action or update). This instruction might also be called a *message* (particularly when the model is a reducer, as described below). The transformation of view actions into model actions and other logic along this path is called *interaction logic*.

A *model update* is a change to the state of one or more of the model objects. When we write *model notification*, we mean an observable notification coming from the model layer, which describes what has changed in the model layer. When views are dependent on model data, notifications should trigger a *view change* to update the contents of the view layer. The transformation of model notifications and data into view changes and other logic along this path is called *presentation logic*. The notifications can take multiple forms: Foundation’s Notification, delegates, callbacks, or another mechanism.

Depending on the application pattern, some state may be maintained outside the model and therefore actions that update that state do not follow this path. A common example of this in many patterns is the *navigation state* where subsections of the view hierarchy (called *scenes*, following the terminology used by Cocoa Storyboards) may be swapped in and out.

The state of an app that is not part of the model is called *view-state*. In Cocoa, most view objects manage their own view-state, and controller objects manage the remaining view-state. Diagrams of view-state in Cocoa typically involve shortcuts across the feedback-loop or individual layers that loop back on themselves.

When all state is maintained in the model layer and all changes follow this full feedback loop path, we call it *unidirectional data flow*. If the only way for any view object or intermediate layer object to be created or updated is via notifications from the model (there is no shortcut where a view or intermediate layer can update itself or another view), the pattern is usually unidirectional.

Architectural Technologies

The standard Cocoa frameworks on Apple's platforms provide some architectural tools. *Notifications* broadcast values from a single source to zero or more listeners. *Key-value observing* can report changes to a property on one object to another object. However, the list of architectural tools in Cocoa runs out quickly.

One of the third-party technologies used in this book will be reactive programming. *Reactive programming* is another tool for communicating changes but unlike notifications or key-value observing, it focuses on the transformations between the source and destination, allowing logic to be expressed in-transit between components.

If a change communication technique like reactive programming or key-value observing is used to communicate changes from property or output on a source object, directly to a property or input on a target object, we call this change communication a *binding*. Cocoa on macOS includes Cocoa Bindings which are a two-way form of binding – all observables are also observers and establishing a binding connection in one direction also establishes a connection in the reverse direction. None of the bindings provided by RxCocoa or CwIViews are two-way — so when we discuss bindings in this book, it will refer solely to one-way bindings.

Application Design Patterns

For the program to function, views must be created, populated with model data, configured to make changes on the model and updated when the model updates.

For this reason, an application requires decisions to be made about how to perform the following tasks:

1. **Construction:** who constructs the model, the views and connects the two?
2. **Updating the model:** how are view actions handled?
3. **Changing the view:** how is model data applied to the view?
4. **View-state:** how is navigation and other non-model state handled?
5. **Testing:** what testing strategies are used to achieve reasonable test-case code coverage?

The answers to these five questions form the basis of the application design patterns we'll look at in this book.

Overview of Application Design Patterns

2

Warning: *This is a pre-release, the final contents will change. The text isn't copy-edited yet, we apologize for spelling and grammatical errors.*

This book will focus on six different implementations of the “Recordings” application, using the following application architectural patterns.

The first three are patterns in common use on iOS:

- **Standard Cocoa Model-View-Controller** (MVC) is the pattern used by Apple in its sample applications. This is easily the most common architecture in Cocoa applications, and the baseline for any discussion about architecture in Cocoa.
- **Online-Only Model-View-Controller** (OO-MVC). Very similar to MVC, but without an explicit model layer: this architecture only displays data coming directly from the server.
- **Model-View-ViewModel+Coordinator** (MVVM-C). A variant of MVC with a separate view-model, and a coordinator to manage the view-controller hierarchy. MVVM uses data binding (typically with reactive programming) to establish the connections between the view-model and the view layer.

The other three patterns we want to look at are experimental architectures that are uncommon in Cocoa. We believe they offer useful insights into application architecture that can help improve your code, regardless of whether you adopt the entire architecture.

- **The Elm Architecture** (TEA) is the most radical departure. You use a virtual view hierarchy to construct views, and reducers to interact between models and views.
- **Model-View-Controller+ViewState** (MVC+VS) centralizes the entire view-state into a single location, following the same rules as the model, rather than spread among views and view-controllers.
- **ModelAdapter-ViewBinder** (MAVB) is an experimental architecture pattern by one of the authors. MAVB focuses on declarative view construction and uses bindings rather than controllers to communicate between model and view.

In this chapter, we'll give an overview of the philosophy and choices behind each of these patterns before subsequent chapters look at the impacts of those choices on the implementation of the “Recordings” app.

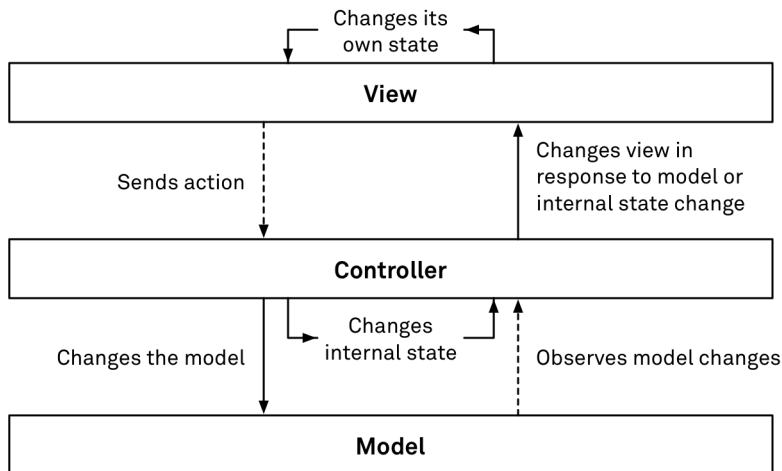
These six patterns are certainly not an exhaustive list of application design patterns for iOS. As we discuss each pattern, we'll try to discuss why we think each of these patterns is worth discussing in the book. At the end of this chapter, we'll briefly discuss some of the patterns that we've omitted.

Model-View-Controller

In Cocoa Model-View-Controller (MVC), a small number of controller objects handle all tasks that fall outside the model or view layers.

This means that the controller layer receives all view actions, handles all interaction logic, dispatches all model actions, receives all model notifications, prepares all data for presentation and applies changes to the views. If you look at the diagram of the application feedback loop in the Introduction chapter, the controller is every labelled point on both arcs between model and view, plus construction and navigation tasks that aren't labelled.

The following is a block diagram of MVC, showing the major communication paths through an MVC application:



The dotted lines in this diagram represent runtime references; neither the view layer nor the model layer reference the controller layer directly in code. Solid lines represent compile-time references; a controller instance knows the interface of the view and model objects to which it's connected.

If you trace the boundary on the outside of this diagram, you get the MVC version of the application feedback loop. Of course, there are other paths through the diagram that don't take this whole path (indicated by the arrows that curve back on the view and controller layers).

Importance of MVC

As the pattern used by Apple in all sample applications – and the pattern which Cocoa is designed to support – Cocoa Model-View-Controller is the authoritative default application architectural pattern on iOS, macOS, tvOS and watchOS.

The specific implementation of the Recordings app presented in this book offers an interpretation of Model-View-Controller that we feel most accurately reflects a common denominator across the history of iOS and macOS. As you'll see in the second part of this book though, the openness of the Model-View-Controller pattern permits a large number of variants and permits other patterns to be integrated within smaller sections of the overall app.

Construction

The application object starts construction of top-level view-controllers. These load and configure views and include knowledge about which data from the model must be represented. A controller will either explicitly create and own the model layer or it will attempt to access the model through a lazily constructed singleton, causing construction with global lifetime. In multi-document arrangements, the model layer is owned by a lower-level controller like `UIDocument/NSDocument`. Cached references to individual model objects relevant to views are usually held by the controllers.

Updating the Model

In MVC, the controller receives view events mostly through the target/action mechanism and delegates (set up in either storyboards or code). The controller knows what kind of views it's connected to, but the view has no static knowledge of the controller's interface. When a view event arrives, the controller can change its internal state, change the model, or directly change the view hierarchy.

Changing the View

In our interpretation, the controller should not directly change the view hierarchy when a model-changing view action occurs. Instead, the controller is subscribed to model notifications, and changes the view hierarchy once a model notification arrives. This way, the data flows in a single direction: view actions get turned into model changes, and the model sends notifications which get turned into view changes.

View-State

View-state may be stored in properties on the view or the controller as needed. View actions affecting state on a view or controller are not required to pass via the model. View-state can be persisted using a combination of support from the storyboard layer – which can record the active controller hierarchy – and implementation of `UIStateRestoring`, which can be used to read data from the controllers and views.

Testing

One of the defining aspects of MVC from a testing perspective is that there are no cleanly separated interfaces. This makes unit and interface tests impossible, leaving integration tests as the primary testing option.

To do this involves building connected sections of the view and model layers and manipulating either the view or the model half and attempting to read the result in the other.

Even in this approach, there are actions that cannot be programmatically triggered in Cocoa views and states that cannot be programmatically read (requiring UI tests to verify) but test coverage over 80% in view controllers is usually possible.

History of Model-View-Controller

The name Model-View-Controller was first used in 1979 by Trygve Reenskaug to describe the existing application “template pattern” in Smalltalk-76, following a terminology discussion with Adele Goldberg (previous names included Model-View-Editor and Model-View-Tool-Editor).

In the original formulation, views were directly “attached” to model objects (observing all changes) and the purpose of a controller was merely to capture user events and forward them to the model. Both of these traits are products of how Smalltalk worked and have little purpose in modern application frameworks. The original formulation is rarely used.

The enduring concept from the original Smalltalk implementation of Model-View-Controller is the premise of *separated presentation* – that the view and model layers should be kept apart – and the premise that there’s a strong need for a supporting object to aid the communication between the two.

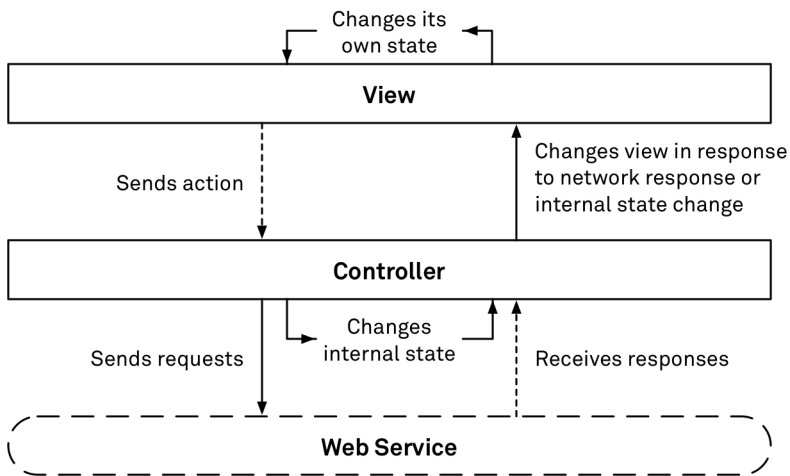
The Cocoa implementation of Model-View-Controller dates back to NeXTStep 4 from approximately 1997. Prior to that time, all of the roles now handled by the controller were typically handled by a high-level view class – often the `NSWindow`. The controller concept in NeXTStep is very similar to the presenter class from Taligent’s earlier Model-View-Presenter. In modern contexts, the name Model-View-Presenter is often used for MVC-like patterns where the view is abstracted from the controller by a protocol.

Online-Only Model-View-Controller

An Online-Only Model-View-Controller (OO-MVC) application has no local storage for primary model data and instead uses network requests to access model data from a web-service as needed by views.

View-controllers are given the additional responsibilities of dispatching requests for data and caching the result in a property of the view-controller. Since the view-controller is the sole owner of its cached model data, this pattern usually forgoes the observer pattern; instead the view-controller manually dispatches changes to views when a new network request completes.

The block diagram looks very similar to MVC:



Importance of OO-MVC

This pattern is prevalent in small iOS apps. Equivalent versions of MVVM with an online-only model also exist and can be considered in similar terms.

Alone of the patterns in this book, Online-Only Model-View-Controller is a pattern that we don't necessarily endorse. We include it due to the importance of networking in many apps and because it gives insight into the effects of over-assignment of responsibility to the controller layer. The second half of the book will include a number of alternative arrangements for this type of application that can more cleanly manage responsibilities.

Construction

This is largely unchanged from MVC; however, on initial construction of each view-controller, the controller object will trigger a network request for data. Upon successful completion, the fetched slice of the model layer will be cached in a property on the controller.

Updating the Model

Views target an action method on their enclosing controller which sends network requests to the server.

Changing the View

When a network request returns, the controller updates its cached model slice and updates the view. Typically, the controller offers a way for the user to explicitly reload the current data, or is subscribed to the server's changes through an open network connection.

View-State

This is unchanged from Model-View-Controller. View-state is implicit in the views and controllers.

Testing

Testing view-state is the same in this pattern as in standard MVC. However, as with all online software, the ability to replace network communication with stubbed calls is essential.

Model-View-ViewModel+Coordinator

Model-View-ViewModel (MVVM), like MVC, is structured around the idea of a scene – a subtree of the view hierarchy that may be swapped in or out during a navigation change.

The defining aspect of MVVM, relative to MVC, is that it uses a *view-model* for each scene to describe the presentation and interaction logic of the scene.

A view-model is an object that does not include compile-time references to the views or controllers but exposes properties that describe the presentation values of the views (the values each view displays) by loading and transforming the underlying model objects for the scene into a format that can be directly set on views. The actual setting of values on views is handled by bindings which take these presentation values and ensure they are set on the views whenever they change.

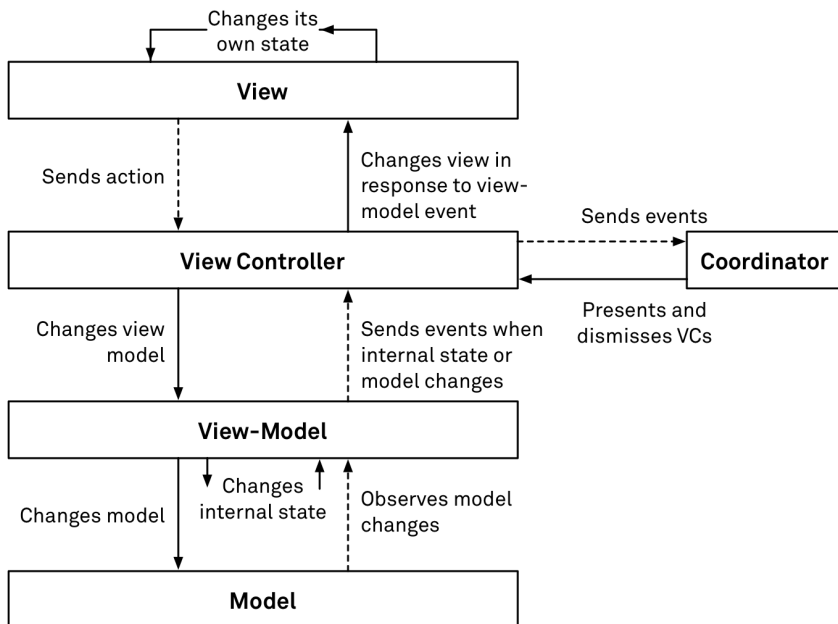
In many cases, the entire view-model can be expressed declaratively using reactive programming bindings. The purpose of a view-model is to express the relationship between the underlying model object, combined with any view-state, and the presentation values displayed in the view. Reactive programming is fundamentally a tool for expressing this type of declarative, transformative relationship, so it's a natural way to express the logic.

Not having a reference to the view layer theoretically makes the view-model independent of the application framework, and allows testing independent from the application framework.

Since the view-model is coupled to the scene, it is helpful to have an object that provides logic between scenes. In MVVM-C, this object is called a *coordinator*.

A *coordinator* is an object which holds references to the model layer and understands the structure of the view-controller tree, so it can provide the required model objects to the view-model for each scene. Neither the view-controller nor the view-model need directly reference other view-controllers or view-models (as happens in MVC during scene transitions). Instead, view-controllers notify the coordinator (through a delegate mechanism) about relevant view actions and the coordinator presents new view-controllers and sets their model data. In other words: the view-controller hierarchy is managed by the coordinator, not by view-controllers.

The resulting architecture has the following overall structure:



If we ignore the coordinator, this diagram is similar to Model-View-Controller – with an additional stage between the view-controller and the model. In many cases this is true, MVVM works like MVC but offloads most of the work from the view-controller onto the view-model which expresses the logic more declaratively and in a way that's abstracted from the views.

For many people, this is the most important point: the view-model has no compile-time reference (solid line) in the direction of the view-controller. The view-model can be separated from the view-controller and views and tested independently. Likewise, the view-controller has no internal view-state anymore – this is moved to the view-model. The double role of the view-controller in MVC (as part of the view-hierarchy and also mediating view to model interactions) is reduced to a single role (the view-controller is solely part of the view-hierarchy).

The coordinators pattern is additive and removes yet another responsibility of view-controllers: presenting other view-controllers. Therefore, it reduces coupling between view-controllers, at the expense of at least one more controller-layer interface.

Importance of MVVM+C

MVVM is the most popular application design pattern on iOS that is not a direct variant of MVC. Though it's not a direct variant of MVC, it's not dramatically different, either; both are structured around view-controller scenes and use most of the same machinery.

The biggest difference is probably the use of reactive programming to express logic in the view-model as a series of transformations and dependencies. Reactive programming and expressing logic through declarative code and understanding how to structure a view-model to clearly describe the relationship between model object and presentation values is an important lesson in understanding dependencies more generally in applications.

Coordinators are a useful pattern in iOS where navigation between scenes is such an important concept. They are not inherently an MVVM concept and can be used with MVC instead but MVVM makes an effort to better express data dependencies within a scene; it makes sense to have a component which handles data dependencies *between* scenes.

Construction

Constructing the model is unchanged from Model-View-Controller and typically remains the responsibility of one of the top level controllers. The individual model objects, however, are owned by the view-models and not by the view-controllers.

The initial construction of the view hierarchy works like MVC, and is done through storyboards or in code. Unlike MVC, the view-controller doesn't directly fetch and prepare data for each view, but leaves this task to the view-model. The view-controller creates a view-model upon construction, and then binds each view to the relevant observable exposed by the view-model.

Updating the Model

In MVVM, the view-controller receives view events in the same way as MVC (and the connection between view and view-controller is set up the same way). However, when a view event arrives, the view-controller doesn't change its internal state, the view-state,

or the model – instead, it immediately calls a method on the view-model. The view-model in turn changes its internal state or the model.

Changing the View

Unlike MVC, the view-controller doesn't observe the model: instead the view-model observes the model and transforms the model notifications in such a way that the view-controller will understand. The view-controller is subscribed to the view-model's changes. Typically, the view-model is observed using bindings from a reactive programming framework, but it could be any observation mechanism. When a view-model event arrives, the view-controller changes the view hierarchy.

To be unidirectional, the view-model should always send model-changing view actions through the model, and only notify the relevant observers after the model change has happened.

View-State

The view-state is either in the views themselves, or in the view-model. Unlike MVC, the view-controller doesn't have any view-state. Changes to the view-model's view-state are observed by the controller, although the controller can't distinguish between model notifications and view-state change notifications. When using coordinators, the view-controller hierarchy is managed by the coordinator(s).

Testing

One of the key advantages that MVVM offers for testing is that the view-model is uncoupled from the view layer yet theoretically represents its state and logic. This allows for testing of the view-model instead of the view controller in most cases.

This does place a burden on the view controller to be foolproof. If testing moves away from the view controller, its logic will need to be far simpler as a consequence.

There is also the problem that view-models are coupled to the view controller. Responsibilities that span between view controllers – navigation and initial construction

– must be handled at the controller, which offer behaves more like a controller than a view-model.

History of Model-View-ViewModel

Model-View-ViewModel was formulated by Ken Cooper and Ted Peters, working at Microsoft on what would become the Windows Presentation Foundation (WPF), an application framework for Microsoft .NET released in 2005.

The Windows Presentation Framework uses a declarative XML-based language called XAML to describe the properties on the view-model to which the view binds. In Cocoa, without XAML, frameworks like RxSwift along with code (usually in the Controller) must be used to perform the binding between view-model and the view.

Model-View-ViewModel is largely the same pattern as the earlier described Model-View-PresentationModel but, as Cooper and Peters described it, Model-View-ViewModel requires explicit framework support for the binding between the view and the view-model whereas PresentationModels traditionally propagate changes manually between the two.

Coordinators in iOS were recently (re)popularized by Soroush Khanlou who described them on his website in 2015, though they are based on the older pattern of Application Controllers that have been in Cocoa and other platforms for decades.

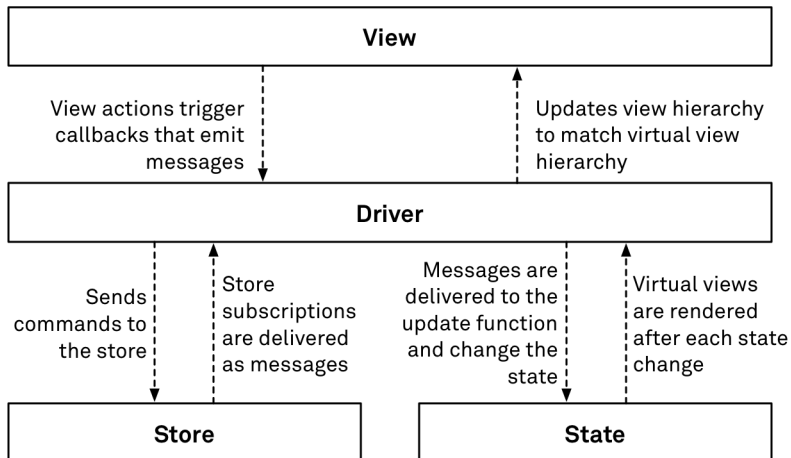
The Elm Architecture

The Elm Architecture (TEA) is a more radical departure from MVC. In TEA, the model and all view-state are integrated as a single state object and all changes in the application occur by sending “messages” to this state object, which processes the messages in a state updating function called a “reducer”.

In TEA, each change to the state creates a new “virtual view hierarchy”, which is composed of lightweight structs that describe what the view hierarchy should look like.

The Driver (which holds references to the other layers in TEA) compares the virtual to the UIView hierarchy and applies the necessary changes to make the views match their virtual counterparts.

A block diagram of changes in TEA looks like this:



If you trace around the upper part of this diagram, you get the feedback loop between the view and the model that we showed at the start of this chapter; it is a loop from view to state (through the driver) and then back to the view. An extra branch takes the commands from the state updates, and applies them to the Store. The driver picks up changes in the store and informs the state about them by sending a message.

The way events flow in TEA is called “unidirectional data flow” and while TEA didn’t invent the term, it was already using a unidirectional data flow pattern when the term was coined to describe Facebook Flux in 2014.

From just the diagram above, the driver might look like a controller in MVC, but it’s far from it: the driver is a component of the Elm framework that you instantiate once for your application. The driver doesn’t know about the specifics of the application itself. Instead, you pass this information into its initializer: the app’s state, a function to update the state from a message, and a function to render the virtual view hierarchy.

TEA also has a different way of dealing with side-effects (like writing data to disk). When processing a message in the state's update method, you can return a command. These commands are executed by the driver. In our case, the most important commands are for changing the contents of the store. The store in turn is observed by subscriptions owned by the driver. These subscriptions can trigger messages to change the state, which triggers view re-rendering in response.

Importance of The Elm Architecture

The Elm Architecture was first implemented in Elm, a functional programming language. The Elm Architecture is therefore a look into how GUI programming can be expressed in a functional way. TEA is also the oldest “unidirectional data flow” architecture.

Construction

The state is constructed on startup and passed to the runtime system (the driver). The runtime system owns the state. The store is a singleton.

The initial view hierarchy is constructed through the same path as later on: a virtual view hierarchy is computed from the current state and the runtime system takes care of updating the real view hierarchy to match the virtual view hierarchy.

Model Updates

Virtual views have actions associated with them, which get sent when a view event happens. The driver receives these actions and uses the update method to change the state. From the update method you can return a command that describes a change you want to make to the data in the store. The driver interprets this command and executes it. The Elm framework makes it impossible for the views to change the state or store directly.

View Changes

The runtime system takes care of this. The only way to change the views is by changing the state. Therefore, there is no difference between initial construction and updating the view hierarchy.

View-State

The view-state is included in the state. As the view is computed directly from the state, the navigation and interaction state is automatically updated as well.

Testing

With an Elm app, the state for the entire application resides in a single location and fully represents the structural state of the view-hierarchy. Theoretically, this makes testing the application equivalent to testing the state like a model interface. However, triggering actions may be more difficult since specifying the view objects to which the action applies can be made more difficult by the fact that the driver typically hides all references to the view objects.

History of The Elm Architecture

The Elm language is a functional programming language, initially designed by Evan Czaplicki, designed for building front-end web apps. The Elm Architecture is a pattern attributed to the Elm community that reportedly emerged naturally from the constraints of the language and the target environment. The ideas behind it have influenced other web-based frameworks like React, Redux, Flux, etc. There is no authoritative implementation of Elm in Swift but there are a number of research projects. For this book, we have built our own interpretation of the pattern in Swift, developed by Chris Eidhof in 2017. While our specific implementation is not “production-ready”, many of the ideas can be used in production code.

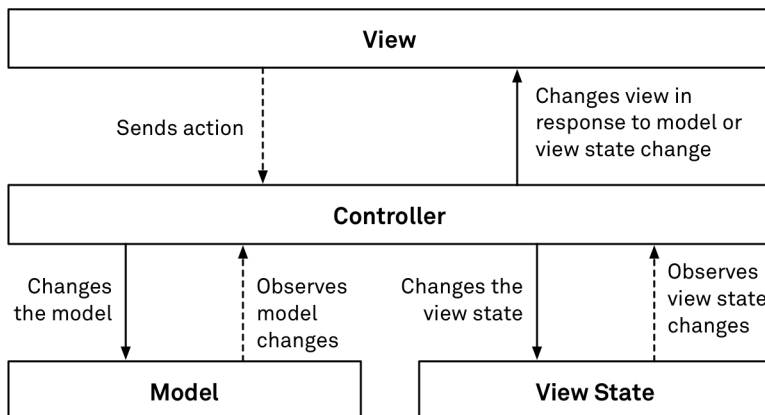
In our description we have used *state*, in Elm this is called *model*. We chose to call it *state* as the word model has a different meaning in this book.

Model-View-Controller + ViewState (MVC+VS)

Model-View-Controller + ViewState (MVC+VS) is an attempt to bring a unidirectional data flow approach to otherwise standard MVC. The aim is to make handling of view-state more manageable, rather than following the two or three different paths that it can take in standard Cocoa MVC.

To make this happen, all view-state is identified and given a representation in a new model object called the view-state model.

Any navigation change, row selection, text field edit, toggle, modal presentation, scroll position change and other view-state must no longer be ignored but must instead be observed by the view-controller and sent to a view-state model action. If these values are needed for presentation or interaction logic, they must not be read from the views but must instead be read from the cached value of the view-state model (the view-state model must be observed by each view-controller and relevant sub-sections cached in the view-controller).



The resulting pattern has some structural similarity to The Elm Architecture, with separate view-state and model, minus the need for a driver, virtual views, messages or reducers.

Of course, the difference here is that the controller is written new for every application. Theoretically, The Elm Architecture driver is written once and never written again.

Importance of MVC+ViewState

MVC+ViewState is primarily a teaching tool for the concept of view-state.

In an application that is otherwise standard MVC, for the simple cost of adding a view-state model and an extra observing function on each view-controller (adding to the model observing already present), this pattern offers arbitrary state restoration (not reliant on storyboards or `UIStateRestoration`), full user-interface state logging and the ability to jump between different view-states for debugging purposes.

Construction

While it's still the responsibility of the view-controllers to apply model data to the views – as in typical MVC – the view-controller is expected to robustly *subscribe* to model data, not merely have a model object set on construction.

This forces a larger amount of work through the notification observing functions – and since there's both a view-state and a model to observe, there are more observing functions than in typical MVC.

Model Updates

When a view action happens, the view-controller either changes the model (unchanged from MVC) or the view-state. The view hierarchy is not changed directly, instead, all changes flow through the model and/or view-state.

View Changes

The controller observes both the model and the view-state, and updates the view hierarchy when changes occur.

View-State

View-state is made explicit, and extracted from the view-controller. The treatment is identical to the model: the controllers observe the view-state, and change the view hierarchy accordingly.

Testing

As with Elm, testing the sum of model and view-state should be equivalent to testing the entire program. Unfortunately, that's not entirely true. Since ensuring view-state is correctly reflected in the view hierarchy is an application responsibility – not a framework responsibility as in Elm – this means that some integration testing is still required to substantially cover this pattern.

History of ViewState-Driven Model-View-Controller

This specific formulation was developed by Matt Gallagher in 2017 as a teaching tool for the concepts of “unidirectional data flow” and “time travel in user-interfaces”. The goal was to make the minimum number of changes required to a traditional Cocoa MVC app so that a snapshot could be taken of the state of the views on every action.

ModelAdapter–ViewBinder

The ModelAdapter-ViewBinder (MAVB) is an experimental pattern best described by example. Consider the following text description of a simple view containing a text field backed by a model object and a checkbox backed by a view-state object:

My View

-- has a text field

-- on text did change ---- send text to here ----> -- name (input)

-- text value <--- get value from here --- -- name (output)

-- has a checkbox

My View-state

-- on toggle -- toggle value over here -> -- selected (input)

-- isChecked <--- get value from here --- -- selected (output)



The premise of ModelAdapter-ViewBinder is that this text description shows all of the necessary user-facing parts of an application design pattern and simultaneously shows the approximate ideal syntax for constructing all the objects involved.

You should be able to see that the pairs of bindings in the middle column form an application feedback loop between the ViewBinder and ModelAdapter columns of the diagram.

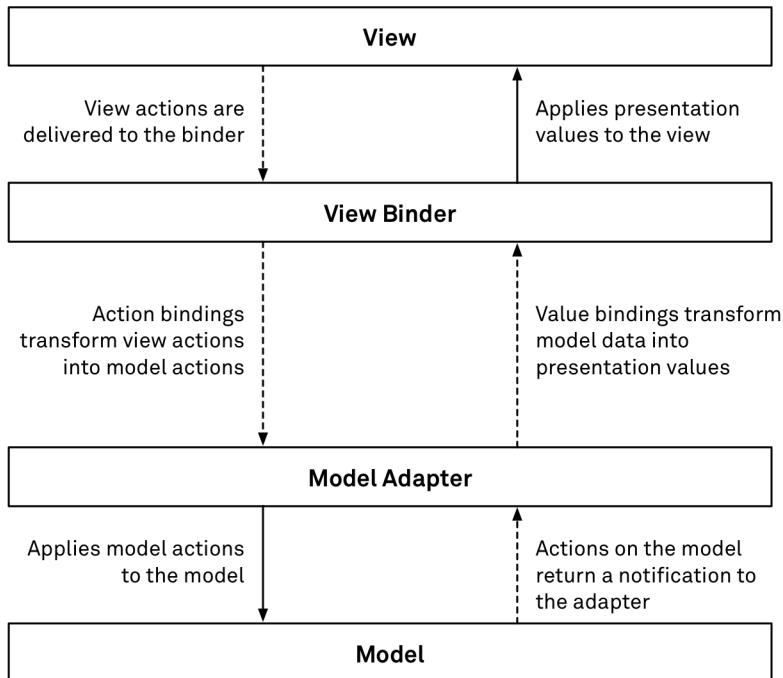
The middle column shows the bindings. These are handled by a reactive programming library, as in MVVM, and are capable of typical reactive programming combining and transformations.

The left column shows the view-binder. A view-binder fully describes a view by the properties that need to be statically configured or dynamically bound. The view-binder is the view's constructor and lazily constructs the view when needed.

The right column shows the model-adapter. A model-adapter is an interface over a state object that exposes an input and output binding and updates the internal state using a reducer function.

The actual view and the model are *not* shown in this diagram. Mutable state is never exposed in MAVB, so you're encouraged to work with the view-binder and model-adapter as though they were the view and model (unless you're writing custom binders or adapters).

The full diagram for the pattern is:



There is no controller layer in this architecture. There is no single pipeline that owns everything else. These control and management roles are entirely fulfilled on a per-property basis by bindings.

Importance of MAVB

Alone of the major patterns we're examining, MAVB has no direct precedent – it is not an implementation of a pattern from another platform or a variation on another pattern. It's its own thing; experimental and a little weird. Its inclusion here is to show something that's proudly different.

That's not to say it takes no lessons or ideas from other patterns. Bindings, reactive programming, domain-specific-language like syntax within a language and reducers are all well known ideas.

Construction

A model-adapter (wrapping the main model) and a view-state-adapter (wrapping the top-level view-state) are typically constructed in the `main.swift` file before any views.

View-binders are constructed on the stack in plain functions, passing in any model-adapters they might require for their bindings. The actual Cocoa views are constructed by the view-binders when you invoke `instance` on them although this happens internally when used by another binding so you don't normally need to do this yourself.

Model Updates

Action bindings are those bindings that carry data emitted from views. Action bindings must connect to model-adapter inputs. When the binding sends a message to the model-adapter, the model-adapter's reducer applies the message to the underlying model.

View Changes

The model-adapter's output binding sends notifications after applying a message. If the model-adapter output is connected to a value binding (a value binding is one that applies a value to the view) then the view will be updated by the binding.

View-State

View-state is considered a part of the model layer. View-state actions and view-state notifications take the same path as model actions and model notifications.

Testing

Testing involves creating a controller set of parameters to a view construction function, calling the view construction function and then decomposing the returned view-binder into its constituent bindings and testing that changes to the input parameters are correctly reflected in the bindings.

This testing of view-binders is structurally similar to the testing of view-models in MVVM. It has the advantage that view-binders span between scenes (so separate Coordinator testing is not required) and the framework applies bindings to the views (rather than relying on the user-code in the view controller as in MVVM) so there are fewer responsibilities that must be tested through UI tests.

History of ModelAdapter-ViewBinder

ModelAdapter-ViewBinder was first described by Matt Gallagher on the Cocoa with Love website. The pattern draws inspiration from Cocoa Bindings, Functional Reactive Animation, ComponentKit, XAML, Redux and experiences with multi-thousand line Cocoa view-controllers.

The specific implementation in this book uses the CwViews framework to handle the view construction, binder and adapter implementations.

Patterns Not Covered

We've chosen to cover six different patterns in this book. Three of them are MVC variants. At least two (MAVB, TEA) are experimental or not production ready.

Why these patterns and not some of the other design patterns that exist in shipping applications?

Model-View-Presenter (MVP)

This is a pattern that's popular on Android and has implementations on iOS. It sits roughly between standard MVC and MVVM in terms of overall structure and technologies used.

MVP uses a separate presenter object that occupies the same position that the view-model occupies in MVVM. Relative to the view-model, the presenter omits reactive programming and omits the expectation that presentation values are exposed as properties on the interface; instead when values need to change, the presenter immediately pushes them down to the view-controller (which is exposed to the

presenter as a protocol) and the the view-controller immediately applies them to the view.

In terms of abstractions, MVP offers nothing over MVC. The reality is that Cocoa's Model-View-Controller, despite its name, *is* MVP – it is derived from Taligent's original Model-View-Presenter implementation in the 1990s. Views, state and related logic are represented identically in both patterns. The only difference is that modern MVP uses a protocol boundary between the presenter and the view whereas Cocoa MVC lets the controller directly reference the view.

Some developers believe the protocol separation is necessary for testing. When we discuss testing, we'll show how standard MVC can be fully tested without any separation and we therefore feel that this pattern isn't different enough. If you strongly desire a fully decoupled representation for testing, we think MVVM's approach is simpler: have the view-controller *pull* the values from the view-model via observing rather than having the presenter push values to a protocol.

VIPER, Riblets and Other “Clean” Patterns.

These patterns are efforts to bring Bob Martin's “Clean Architecture” from web applications to iOS development by spreading the roles of the controller across three to four different classes with a strict sequential ordering. Each class is forbidden from directly referencing preceding classes in the sequence.

To ensure the rules about referencing in one direction only, these patterns require a *lot* of protocols, classes and passing data between layers. For this reason, many developers using these patterns use code generators. Our feeling is that code generators – and any patterns verbose enough to imply their need – are misguided. Attempts to bring “clean architecture” to Cocoa usually claim to manage “massive view controllers” but, ironically, do so by making the codebase even larger.

While we feel that interface decomposition is valid approach for managing code size, we feel it should be performed as-needed, rather than methodically and per view-controller. Decomposition should be performed along with knowledge of the data and tasks involved so that the best abstractions – and hence the best reduction in complexity – can be achieved.

Component-Based Architecture (React Native)

If you prefer programming in Javascript to Swift or your application relies heavily on web API interactions that work better in Javascript then perhaps you should consider React Native. However, this book is focused on Swift and Cocoa, so we're limiting the patterns explored to those domains.

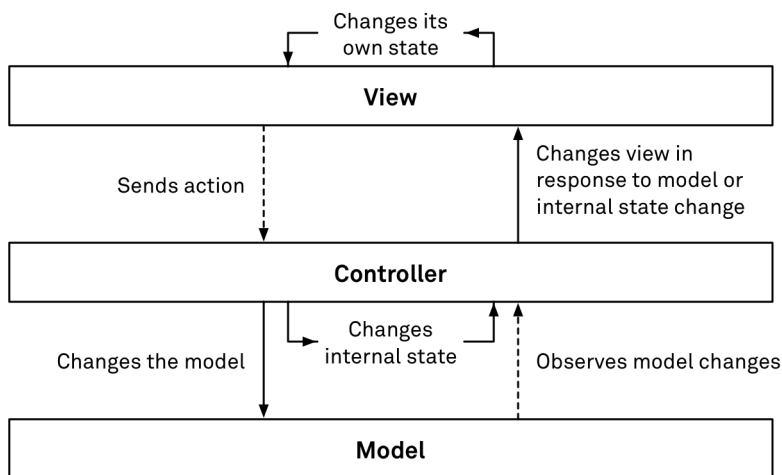
If you're looking for something like CBA or React Native, then take a look at our exploration of The Elm Architecture, an antecedent to Redux which is commonly used with React Native to handle unidirectional data flow. The ModelAdapter-ViewBinder implementation also takes inspiration from ComponentKit – itself React-inspired – and uses a DSL-like syntax for declarative, transformational view construction with parallels to the render function in React Components.

Model View Controller

3

The Model-View-Controller pattern (MVC) is the baseline for all other patterns discussed in this book. It is the authoritative pattern of the Cocoa frameworks and the oldest pattern that we will discuss in this book.

The key premise of Model-View-Controller is that the model layer and view layers are brought together by the controller layer, which constructs and configures the other two layers and mediates communication – in both directions – between model and view objects. Thus the controller layer represents the backbone on which the application feedback loop is formed in MVC apps.



Model-View-Controller is based around classic object-oriented principles: objects manage their behaviors and state internally and communicate via class and protocol interfaces. View classes are typically self-contained, reusable objects; model objects are abstract and avoid dependence upon the rest of the program; it is therefore the responsibility of the controller layer to pull these other layers together and give them their purpose and behavior within the greater context of the program.

Apple describes MVC as a collection of three distinct sub-patterns:

1. Composite pattern – views are assembled together into hierarchies and sections of the hierarchy are managed in groups by controller objects

2. Strategy pattern – controller objects are mediators between the view and model and manage all application-specific behaviors for reusable, application-independent views
3. Observer pattern – objects dependent on model data must subscribe to receive updates

The MVC pattern is often interpreted quite loosely such that these sub-patterns – in particular, the observer pattern – are not always followed. We will include all three as part of our implementation.

The Model-View-Controller pattern has some well known deficiencies, foremost of which is the *massive view controller* problem (using the same MVC acronym as the pattern) where the controller layer takes on too many responsibilities. Model-View-Controller also presents difficulties in testing – making unit tests and interface tests difficult or impossible. Yet despite these shortcomings, Model-View-Controller remains the simplest pattern to use for iOS apps, can scale to suit any program – provided you understand its shortcomings and how to counter them – and can include robust testing.

Exploring The Implementation

Construction

The construction process in Cocoa MVC largely follows the default startup process as provided by the Cocoa framework. The key aim of the default Cocoa application startup process is to ensure the construction of three objects: the UIApplication object, the application delegate, and the root view controller of the main window. The configuration of this process is distributed across three files – named Info.plist, AppDelegate.swift and Main.storyboard, by default.

These three objects – all part of the controller layer – provide configurable locations to continue the startup process and therefore place the controller layer in charge of all construction.

Connecting Views to Initial Data

Views in an MVC app do not directly reference model data; they are kept self-contained and reusable, without knowledge about application-specific details such as model data. Instead, model objects are stored in the view controller. This makes the view controller a non-reusable class but that's the purpose of view controllers: to give application-specific knowledge to other components in the program.

The model object stored in the view controller gives it *identity* (letting the view controller understand its position in the program and how to talk to the model layer). The view controller extracts and transforms the relevant property values from the model object before setting these transformed values on any child views.

Exactly how this identity object is set on a view controller varies. There are two different strategies used in the Recordings app for setting the initial model value on view controllers:

1. Immediately access a model object for the controller based on the controller's type or location in the controller hierarchy.
2. Initially set references to model objects to nil and keep everything in a blank state until a non-nil value is provided by another controller.

A third strategy – passing the model object on construction (also known as dependency injection) – would be preferred if it were possible, however, the Storyboard construction process typically prevents passing construction parameters to view controllers.

The `FolderViewController` (the master view in the top-level split view) is an example of the first strategy. Each folder view controller initially sets its folder property as follows:

```
var folder: Folder = Store.shared.rootFolder
```

This means that on initial construction, every folder view controller assumes it represents the *root* folder. If it actually is a child folder view controller then the parent folder view controller will set the folder value to something else during `perform(segue:)`. This approach ensures that the folder property in the folder view controller is not optional, and the code can be written without need to conditionally test for the existence of the folder object, since there is always at least the root folder.

In the Recordings app, the model object `Store.shared` is a lazily constructed singleton. This means that when the first folder view controller attempts to access the `Store.shared.rootFolder` is probably the point when the shared store instance is constructed.

The `PlayViewController` (the detail view in the top-level split view), uses an initially-nil optional reference for its model object and therefore represents the second strategy for setting the identity reference:

```
var recording: Recording?
```

Unless this value is set by the folder view controller (or state restoration), the play view controller will show a blank (“No recording selected”) display.

In either case, if this primary model object is set, the controller must respond by updating the views.

An example of this in the folder view controller is that the `navigationItem.title` (the display of the folder name in the navigation bar at the top of the screen) must be updated when the folder is set:

```
var folder: Folder = Store.shared.rootFolder {
    didSet {
        tableView.reloadData()
        if folder === Store.shared.rootFolder {
            navigationItem.title = .recordings
        } else {
            navigationItem.title = folder.name
        }
    }
}
```

The call to `tableView.reloadData()` will also ensure that all `UITableViewCell`s displayed are updated too.

As a rule, whenever we read the initial model data, we must also observe changes to the model. In the folder view controller’s `viewDidLoad`, we add the view controller as an observer of model notifications:

```

override func viewDidLoad() {
    super.viewDidLoad()
    // ...
    NotificationCenter.default.addObserver(self, selector: #selector(handleChangeNotification(_:)), name:
}

```

State Restoration

State restoration in MVC uses the Storyboard system, which acts as part of the controller layer. Opting into this system requires the following methods implemented on the AppDelegate:

```

func application(_ application: UIApplication, shouldSaveApplicationState: NSCoder)
-> Bool {
    return true
}

```

```

func application(_ application: UIApplication, shouldRestoreApplicationState: NSCoder)
-> Bool {
    return true
}

```

From that point, the Storyboard system takes over. View controllers that should be saved and restored automatically by the Storyboard system are configured with a Restoration ID. For example, the root view controller in the Recordings app has the Restoration ID `splitController` specified on the Identity Inspector in the Storyboard editor. Similar identifiers exist for every scene in the Recordings app except the `RecordViewController` (which is deliberately non-persistent).

While the Storyboard system will preserve the existence of these view controllers, it will not preserve the model data stored in each view controller without further work. Each `UIViewController` must implement `encodeRestorableState(with:)` and `decodeRestorableState(with:)`. This is the implementation on the `FolderViewController`:

```

override func encodeRestorableState(with coder: NSCoder) {
    super.encodeRestorableState(with: coder)
    coder.encode(folder.uuidPath, forKey: .uuidPathKey)
}

```

The encoding part is simple – the FolderViewController saves the uuidPath that identifies its Folder model object. The decoding part is a little more complicated:

```
override func decodeRestorableState(with coder: NSCoder) {
    super.decodeRestorableState(with: coder)
    if
        let uuidPath = coder.decodeObject(forKey: .uuidPathKey) as? [UUID],
        let folder = Store.shared.item(atUUIDPath: uuidPath) as? Folder
    {
        self.folder = folder
    } else {
        if let index = navigationController?.viewControllers.index(of: self), index != 0 {
            navigationController?.viewControllers.remove(at: index)
        }
    }
}
```

After decoding the uuidPath, the FolderViewController must check that the item still exists in the Store so it can set its self.folder property to the item from the Store.

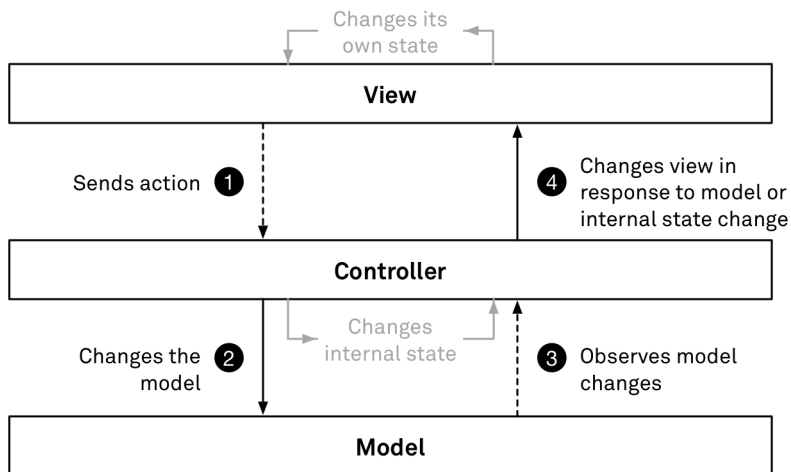
If the item *doesn't* exist in the Store, then the FolderViewController must attempt to remove itself from its parent navigationController's list of view controllers.

Changing the Model

The broadest interpretations of Model-View-Controller don't include any details about how to implement the model, how model changes should occur or how the view should respond to them. In the earliest versions of macOS, following the precedent set by the earlier Document-View patterns, it was common to have controller objects like NSWindowController or NSDocument directly change the model in response to view actions and then directly update the view as part of the same function.

For our implementation of Model-View-Controller, we believe it is essential that actions which update the model must not occur in the same function as changes to the view hierarchy. Actions that update the model must return, without assuming any effect on the model state. After the construction phase, changes to the view hierarchy may occur only as a result of observation callbacks – following the observer pattern that is part of the MVC formulation.

The observer pattern is essential to maintaining a clean separation of model and view in MVC. The advantage of this approach is that we can be sure the UI is in sync with the model data, no matter where the change originated from (e.g. from a view event, a background task, or from the network). Also, the model has a chance to reject or modify the requested change.



In this section, we'll look at the steps involved to delete an item from a folder.

Step 1: Table View Sends Action

In our example app, the `dataSource` of the `UITableView` is set to the `FolderViewController` by the Storyboard. To handle the tap on the delete button, the table view invokes `tableView(_:commit:forRowAt:)` on its `dataSource`.

Step 2: View Controller Changes Model

The implementation of `tableView(_:commit:forRowAt:)` looks up the associated child item (based on the index path) and asks the parent folder to remove it:

```
// FolderViewController.swift
```

```

override func tableView(_ tableView: UITableView, commit editingStyle:
    UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
    folder.remove(folder.contents[indexPath.row])
}

```

Note that we are not deleting the cell from the table view directly. This only happens once we observe the model change.

The remove method on Folder removes the requested item from the folder's contents and notifies the item that it has been removed by calling `item.deleted()`. Then it tells the store to persist the data, including detailed information about the change it just made:

```

// Folder.swift
func remove(_ item: Item) {
    guard let index = contents.index(where: { $0 === item }) else { return }
    contents.remove(at: index)
    item.deleted()
    store?.save(item, userInfo: [Item.changeReasonKey: Item.removed, Item.oldValueKey:
        index, Item.parentFolderKey: self])
}

```

If the deleted item is a recording, the associated file is removed from the filesystem by the call to `item.deleted()`. If it is a folder, it recursively calls `deleted()` on all children to delete all contained sub-folders and recordings.

Persistence of the model objects and sending the change notification occurs in the call to `store?.save`.

```

// Store.swift
func save(_ notifying: Item, userInfo: [AnyHashable: Any]) {
    if let url = baseURL,
        let data = try? JSONEncoder().encode(rootFolder) {
        try! data.write(to: url.appendingPathComponent(.storeLocation))
    }
    NotificationCenter.default.post(name: Store.changedNotification, object: notifying,
        userInfo: userInfo)
}

```

Step 3: View Controller Observes Model Changes

The folder view controller sets up an observation of the store's change notification in `viewDidLoad`:

```
// FolderViewController.swift
override func viewDidLoad() {
    super.viewDidLoad()
    // ...
    NotificationCenter.default.addObserver(self, selector: #selector(handleChangeNotification(_:)), name:
}
```

The call to `save` on the store from the previous step triggers this observation and therefore calls `handleChangeNotification`.

Step 4: View Controller Changes View

When the store change notification arrives, the view controller's `handleChangeNotification` method interprets it, and makes the corresponding change in the view hierarchy.

A minimalist handling of notifications might involve reloading the table data whenever any type of model notification arrives. In general though, correct handling of model notifications involves properly understanding the data change described by the notification. Accordingly, our implementation communicates the nature of the model change through the `userInfo` dictionary – including the specific row that has changed and the type of change that has occurred.

In this example, handling the notification involves a call to `tableView.deleteRows(at:with:)`:

```
// FolderViewController.swift
@objc func handleChangeNotification(_ notification: Notification) {
    // ...
    if let changeReason = userInfo[Item.changeReasonKey] as? String {
        switch (changeReason, userInfo[Item.newValueKey], userInfo[Item.oldValueKey]) {
        case (Item.removed, _, .some(let oldIndex as Int)):
            tableView.deleteRows(at: [IndexPath(row: oldIndex, section: 0)], with: .right)
```

```
// ...  
}  
}  
}
```

Noticeably absent from this code: we have not updated any data on the view controller itself. The `folder` value on the view controller is a shared reference directly to the object in the model layer so it is already up-to-date. After this call to `tableView.deleteRows(at:with:)` the table view will call the `UITableViewDataSource` delegate implementations on the `FolderViewController` and they will return the latest state of the data accessed through the `folder` shared reference.

We should also clarify that this notification handling relies on a short-cut: the model stores `Items` in the same lexically sorted order that is used for display. This is *not* ideal; the model should not really know *how* its data will be displayed. A conceptually cleaner implementation (though more work) would involve the model emitting *set* mutation information (not array mutation information) and the notification handler using its own sorting combined with before and after states to determine deleted indices.

This completes the event loop in MVC. Since we updated the UI only in response to the change in the model (and not directly in response to the view action), the UI updates correctly even if the folder would be removed from the model for other reasons (for example, a network event), or if the change gets rejected by the model. It's a robust approach to make sure the view layer does not get out of sync with the model layer.

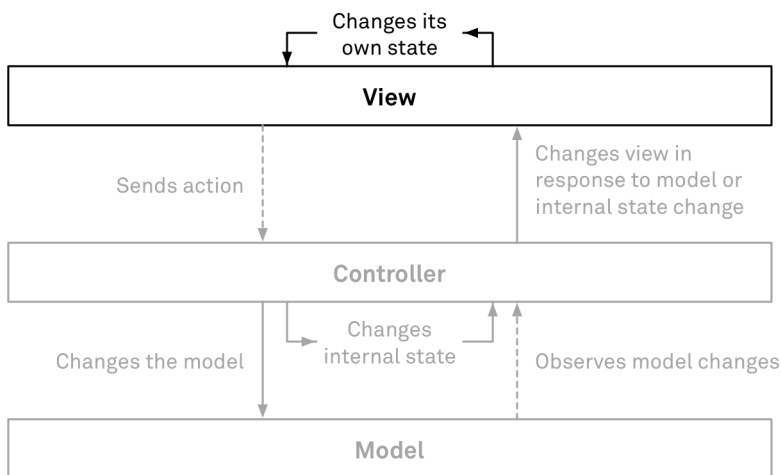
Changing The View-State

The model layer in MVC has its origins in typical document-based applications where any state that would be written to the document during a save operation is considered part of the model. Any other state – including navigation state, temporary search and sort values, feedback from asynchronous tasks and uncommitted edits – is traditionally excluded from the definition of the model in MVC.

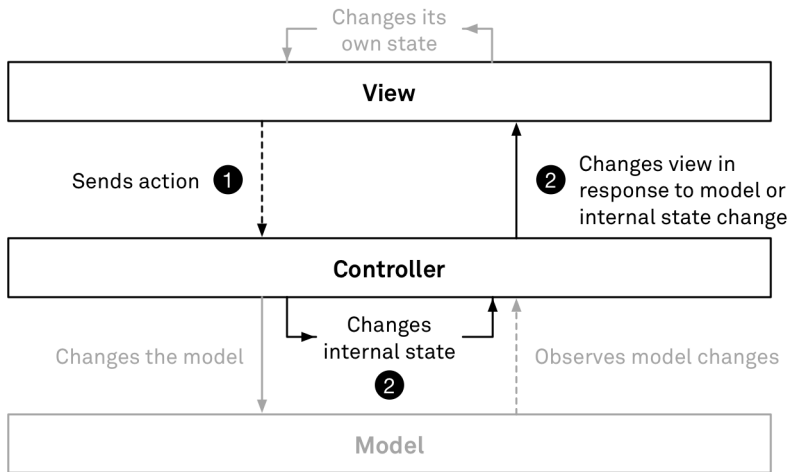
In MVC, this “other” state – which we collectively call view-state – does not have a description in the pattern. In accordance with traditional object-oriented principles, any object may have internal state and the object is not required to communicate changes in that internal state to the remainder of the program.

Due to this internal treatment, view-state does not necessarily follow any clear path through the program. Any view or controller may include state which it updates in response to view actions. View-state is handled as locally as possible: if a view or view controller can autonomously update its view-state in response to a user event, it does so without any further consultation of other parts of the program.

Most UIViews have internal state that they may update in response to view actions, without propagating the change further. For example, a UISwitch may change from ON to OFF in response to a user touch event as follows:



The event loop becomes one step longer if the view cannot change its state itself. For example, when the label of a button should change after it's tapped (as with our play button) or when a new view controller is pushed onto the navigation stack in response to the user tapping a table view cell.



The view-state is still contained within one particular view controller and its views. However, compared to a view changing its own state we now get the chance to customize the state change of a view (as the play button title in the first example below) or to make view-state changes across views (exemplified by the second example of pushing a new folder view controller below).

Example 1: Updating the Play Button

The play button in the play view controller changes its title from “Play” to “Pause” to “Resume”, depending on the play state. When the button says “Play”, tapping it changes it to “Pause”; tapping it again before playback ends changes it to “Resume”. Let’s take a closer look at the steps Cocoa MVC takes to achieve this.

Step 1: The Button Sends an Action to the View Controller

The play button connects to the play view controller’s play method using an `IBAction` in the Storyboard. Tapping the button calls the play method:

```
@IBAction func play() {  
    // ...
```

```
}
```

Step 2: The View Controller Updates its Internal State

The first line in the play method updates the state of the audio player:

```
@IBAction func play() {  
    audioPlayer?.togglePlay()  
    // ...  
}
```

Step 3: The View Controller Updates the Button

The second line in the play method calls `updatePlayButton`, which directly sets the new title of the play button, depending on the state of the audio player:

```
@IBAction func play() {  
    // ...  
    updatePlayButton()  
}  
  
func updatePlayButton() {  
    if audioPlayer?.isPlaying == true {  
        playButton?.setTitle(.pause, for: .normal)  
    } else if audioPlayer?.isPaused == true {  
        playButton?.setTitle(.resume, for: .normal)  
    } else {  
        playButton?.setTitle(.play, for: .normal)  
    }  
}
```

`.pause`, `.resume`, and `.play` are localized strings defined as static constants on `String`.

At this point we're already done. The smallest number of components necessary were involved in the process: the button sends the event to the play view controller, the play view controller in turn sets the new title.

Example 2: Pushing a Folder View Controller

The folder view controller's table view shows two kinds of items: recordings and sub-folders. When the user taps on a sub-folder, a new folder view controller is configured and pushed onto the navigation stack. Since we use a Storyboard with segues to achieve this, the concrete steps below relate only loosely to the steps in the diagram above. However, the general principle stays the same.

Step 1: Triggering the Segue

Tapping a sub-folder cell triggers a `showFolder` segue, as the cell is connected to the folder view controller via a push segue in the Storyboard. This causes UIKit to create a new folder view controller instance for us.

This step is a variant of the target/action pattern. There's more UIKit magic involved behind the scenes, but the result is that the `prepare(for:sender:)` method on the originating view controller gets called .

Step 2 & 3: Configuring the New Folder View Controller

The current folder view controller gets notified about the segue that is about to occur using `prepare(for:sender:)`. After we check the segue identifier, we configure the new folder view controller:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    guard let identifier = segue.identifier else { return }
    if identifier == .showFolder {
        guard
            let folderVC = segue.destination as? FolderViewController,
            let selectedFolder = selectedItem as? Folder
        else { fatalError() }
        folderVC.folder = selectedFolder
    }
    // ...
}
```

First, we check that the destination view controller has the right type and we have selected a folder. If either of these conditions doesn't hold it is a programming error and

we crash. If everything is present as expected, we set the sub-folder on the new folder view controller.

The Storyboard machinery takes over the actual presentation of the new view controller. After we configure the new view controller, UIKit pushes it onto the navigation stack (we don't need to call `pushViewController` ourselves). Pushing the view controller onto the navigation stack causes the navigation controller to install the view controller's view in the view hierarchy.

Similar to the play button example, the UI event (selecting the sub-folder cell) is handled as locally as possible. The segue mechanism obscures the exact path of the event, nevertheless, from the perspective of our code no other component but the originating view controller is involved. The view-state is implicitly represented by the involved views and view controllers.

To share view-state between different parts of the view hierarchy, we must find the common ancestor in the view (controller) hierarchy, and manage the state there. For example, to share the play state between the play button label and the player, we store the state in the play view controller. When you need view-state that's used by almost all components (for example, a boolean value that captures whether your application is in dark mode), you would have to put it in one of the topmost controller objects (e.g. the application delegate). However, it's uncommon in practice to put this view-state in a controller object at the top of the hierarchy, because that requires a communication channel between each layer of the hierarchy, so most people opt to use a singleton instead.

Testing

Automated tests come in a number of different forms. From smallest granularity to largest granularity, these are:

- Unit tests (isolating individual functions and testing their behavior)
- Interface tests (using interface inputs – usually functions – and testing the result via interface outputs – usually different functions)
- Integration tests (testing the program – or significant subsections – as a whole)

While interface and data-driven regression tests have been around for forty years or more, modern unit testing didn't really start until the 1990s and took at least another decade to become common in applications. Even now, many apps have no regular testing outside of human-driven tests.

It should not be surprising then that the 20+ year old Cocoa MVC pattern was created without unit testing in mind. You can test the model however you choose – since it is independent of the rest of the program – but this won't help you test user-facing state. You can use Xcode's UI tests – automated scripts that run your entire program and attempt to read the screen using the VoiceOver/Accessibility API – but this is slow, prone to timing issues and difficult to extract precise results.

If we want to use precise, code-level testing of the controller and view layers in MVC, the only option available are integration tests. Integration tests involve constructing a self-contained version of the app, manipulating part of it and reading from other parts to ensure that results are propagated between objects as expected.

For the Recordings app, the tests require a Store, so we construct one without a URL (in-memory only) and add some test items (folders and recordings):

```
func constructTestingStore() -> Store {
    let store = Store(url: nil)

    let folder1 = Folder(name: "Child 1", uuid: uuid1)
    let folder2 = Folder(name: "Child 2", uuid: uuid2)
    store.rootFolder.add(folder1)
    folder1.add(folder2)

    let recording1 = Recording(name: "Recording 1", uuid: uuid3)
    let recording2 = Recording(name: "Recording 2", uuid: uuid4)
    store.rootFolder.add(recording1)
    folder1.add(recording2)

    store.placeholder = Bundle(for: FolderViewControllerTests.self)
        .url(forResource: "empty", withExtension: "m4a")!

    return store
}
```

The `store.placeholder` is a store feature used for testing only: if the URL is `nil`, this placeholder will be returned any time `Store.fileURL(for:)` is invoked to get the audio file for a recording. With the store constructed, we now need a view controller hierarchy that uses this store:

```
func constructTestingViews(store: Store, navDelegate: UINavigationControllerDelegate) ->
    (AppDelegate, UISplitViewController, UINavigationController, FolderViewController) {
    let storyboard = UIStoryboard(name: "Main", bundle: Bundle.main)

    let navigationController = storyboard.instantiateViewController(withIdentifier:
        "navController") as! UINavigationController
    navigationController.delegate = navDelegate

    let rootFolderViewController = navigationController.viewControllers.first as!
        FolderViewController
    rootFolderViewController.folder = store.rootFolder
    rootFolderViewController.loadViewIfNeeded()

    // ...

    window.isHidden = false
    return (appDelegate, splitViewController, navigationController,
        rootFolderViewController)
}
```

This shows the construction of the master navigation controller and the root view controller. The function goes on to similarly construct the detail navigation controller, the play view controller, split view controller, window and app delegate – a full user-interface stack. Constructing all the way up to the window and then calling `window.isHidden = false` is necessary because otherwise numerous animated and presented actions won't occur.

Notice how the navigation controller's delegate is set to the `navDelegate` parameter (which will be an instance of the `FolderViewControllerTests` class that runs the tests) and the root folder view controller's folder is set to the testing store's `rootFolder`.

We call the above construction functions from the `FolderViewControllerTests.setUp()` function to initialize the members on the `FolderViewControllerTests` instance. With that done, we can write our tests.

Integration tests usually require some degree of configuring the initial environment, before an action is performed and the result is measured. With application integration tests, measuring the result can range from trivial (accessing a readable property on one of the objects in the configured environment) to very difficult (multi-step asynchronous interactions with the Cocoa framework).

An example of a relatively easy test is testing the `commitEditing` action for deleting table view rows:

```
func testCommitEditing() {
    // Verify that the action we will invoke is connected
    let dataSource = rootFolderViewController.tableView.dataSource as? FolderViewController
    XCTAssert(dataSource == rootFolderViewController)

    // Confirm item exists before
    XCTAssert(store.item(atUUIDPath: [store.rootFolder.uuid, uuid3]) != nil)

    // Perform the action
    rootFolderViewController.tableView(rootFolderViewController.tableView,
        commit: .delete, forRowAt: IndexPath(row: 1, section: 0))

    // Assert item is gone afterwards
    XCTAssert(store.item(atUUIDPath: [store.rootFolder.uuid, uuid3]) == nil)
}
```

This tests that the root view controller is correctly configured as the data source, then invokes the data source method `tableView(_:commit:forRowAt:)` directly on the root view controller and confirms that this deletes the item from the model.

When animations or potential simulator-dependent changes are involved, the tests get more complicated. Testing that selecting a recording row in the folder view controller correctly displays that recording in the split view's detail position is the most complex test in the Recordings app. The test needs to handle the difference between the split view controller's collapsed and uncollapsed states, and needs to wait for potential navigation controller push actions to complete:

```
func testSelectedRecording() {
    // Select the row (so `prepare(for:sender:)` can read the selection
    rootFolderViewController.tableView.selectRow(at: IndexPath(row: 1, section: 0),
```



```

        animated: false, scrollPosition: .none)

// Handle collapsed or uncollapsed split view controller
if self.splitViewController.viewControllers.count == 1 {
    ex = expectation(description: "Wait for push")

    // Trigger the transition
    rootFolderViewController.performSegue(withIdentifier: "showPlayer", sender: nil)

    // Wait for the navigation controller to push the collapsed detail view
    waitForExpectations(timeout: 5.0) { e in
        // Ensure we didn't timeout
        XCTAssert(e == nil)

        // Traverse to the `PlayViewController`
        let collapsedNC = self.navigationController.viewControllers.last as?
            UINavigationController
        let playVC = collapsedNC?.viewControllers.last as? PlayViewController
        XCTAssert(playVC != nil)

        // Test the result
        XCTAssert(playVC?.recording?.uuid == uuid3)
    }
} else {
    // Handle the uncollapsed state...
}
}

```

The expectation is fulfilled when the master navigation controller reports that it has showed a new view controller. This change to the master navigation controller happens when the detail view is collapsed onto the master view (as happens in a *compact* display – i.e. every iPhone display except landscape on iPhone Plus).

```

func navigationController(_ navigationController: UINavigationController, didShow
    viewController: UIViewController, animated: Bool) {
    ex?.fulfill()
    ex = nil
}

```

While these integration tests get the job done – the tests we’ve written for the folder view controller cover roughly 80% of its lines and test all important behaviors – this `testSelectedRecording` test reveals that correctly writing integration tests can require significant knowledge about how the Cocoa framework operates.

Discussion

MVC has the advantage that it is the architectural pattern for iOS development with the lowest friction. Every class in Cocoa is tested under MVC conditions. Features like Storyboards that rely on heavy integration between the frameworks and your classes are more likely to work smoothly with your program when you use MVC. You’ll find more examples on the internet that follow an MVC pattern than any other architecture. MVC often has the least code and the least design overhead of all patterns.

As the most used of all the patterns we’ll explore in this book, MVC also has the most clearly understood shortcomings – in particular, MVC is associated with two common problems.

Observer pattern failures

The first of these problems is when the model and view fall out of synchronization. This arises when the observer pattern around the model is imperfectly followed. A common mistake is to read a model value on view construction and not subscribe to subsequent notifications. Another common mistake is to change the view hierarchy at the same time as changing the model – assuming the result of the change instead of waiting for the observation, even when the model might reject the change afterwards. These kinds of mistakes can cause the views to go out of sync with the model, and unexpected behavior follows.

Unfortunately, Cocoa doesn’t provide any checks or built-in mechanisms to verify that you have a correct implementation of the observer pattern. The solution is to be strict in applying the observer pattern: when you read a model value, you also need to subscribe to it. Other architectures (such as TEA or MAVB) combine the initial reading and subscribing into a single definition, making it impossible to make an observation mistake.

Massive view controllers

The second problem associated with MVC – *Cocoa* MVC in particular – is that MVC often leads to large view controllers. View controllers have view layer responsibilities (configuring view properties and presenting views) but they also have controller layer responsibilities (observing the model and updating views) and they can also end up with model layer responsibilities (fetching, transforming or processing data). Combined with their central role in the architecture, this makes it easy to carelessly assign every responsibility to the view controller, rapidly making the program unmanageable.

There's no clear limit for how big a view controller could reasonably be. Xcode starts showing obvious pauses when opening and browsing viewing Swift files over two thousand lines in length so it's probably better to avoid anything near this length. Most screens display fifty lines or less of code so a dozen screens worth of scrolling starts will make it visually difficult to find code.

But the strongest argument against large view controller's isn't the lines of code as much as the amount of state. When the entire file is a single class – like a view controller – any mutable state will be shared across all parts of the file with each function needing to cooperatively read and maintain the state to prevent inconsistency. Separating the maintenance of state into separate interfaces forces better consideration of data dependencies and limits the potential amount of code that must cooperatively obey rules to ensure consistency.

Improvements

Observer pattern

For the base MVC implementation, we chose to use broadcast model notifications using `NotificationCenter`. We chose this because we wanted to write the implementation using basic Cocoa, with limited use of libraries or abstractions.

However, the implementation requires cooperation in many locations to correctly update values. The folder view controller gets its folder value when the parent folder view sets it after creation:

```
let child = storyboard!.instantiateViewController(withIdentifier: .folderController)
```

```
(child as! FolderViewController).folder = f
```

at a later time, the folder view controller observes notifications on the model to get changes to this value:

```
override func viewDidLoad() {  
    // ...  
    NotificationCenter.default.addObserver(self, selector: #selector(handleChangeNotification(_:)), name:  
}
```

and in the notification handler, if the notification matches the current folder, the folder is updated:

```
@objc func handleChangeNotification(_ notification: Notification) {  
    if let item = notification.object as? Folder, item === folder {  
        if notification.userInfo?[Item.changeReasonKey] as? String == Item.removed, let nc = navigationController {  
            nc.setViewControllers(nc.viewControllers.filter { $0 !== self }, animated: false)  
        } else {  
            folder = item  
        }  
    }  
}
```

It would be better to have an observing approach that didn't have the timing gap between the initial setting of folder and the establishment of the observer in `viewDidLoad`. It would also be better if there was only one location where folder needed to be set.

It is possible to use key-value coding instead of notifications but the need to observe multiple different key-paths in most cases (for example, observing the parent folder's children as well as the current child) prevent it from actually making handling more robust. Since it also requires that every observed property be declared dynamic, it is dramatically less popular in Swift than Objective-C.

You can write your own custom delegate protocols but these allow different kinds of callback (not just one-way notifications), the observer pattern rely solely on one-way callbacks, so delegates don't offer many advantages and add the problem of needing to maintain the delegate reference.

The simplest improvement to improve the observer pattern, is to wrap `NotificationCenter` and implement the “initial” concept that key-value observation includes. The “initial” concept sends the initial value as soon as the observation is created, allowing you to combine the concepts of “set initial value” and “observe subsequent values” into a single pipeline.

This allows us to replace the initial construction with:

```
let child = storyboard!.instantiateViewController(withIdentifier: .folderController)
(child as! FolderViewController).uuidPath = f.uuidPath
```

and replace both the `viewDidLoad` and the `handleChangeNotification` with:

```
observations += Store.shared.addObserver(at: uuidPath, type: Folder.self) { [weak self] folder in
    guard let f = folder else {
        navigationController.map {
            $0.setViewControllers($0.viewControllers.filter { $0 != self }, animated: false)
        }
        return
    }
    self.folder = folder
}
```

It’s not dramatically less code but the `self.folder` value is set in just one location (inside the observation callback) so you can reduce the number of change paths in your code. There’s no more need for dynamic casting in user code. There’s no gap between the initial setting of the value and the establishing of the observation, since the initial value set is not the real data but an identifier – the only way we need to access the real data is through the observation callback. We’re also able to include program logic in a common location so the test for whether the `Item.changeReasonKey` is `Item.removed` can happen automatically.

This type of `addObserver` implementation has the advantage that it can be implemented as an extension to the `Store` type, without any changes to the `Store` itself:

```
extension Store {
    func addObserver<T>(at uuidPath: [UUID], type: T.Type, _ callback: @escaping (T?) -> () -> Array<NSObject>
        guard let i = item(atUUIDPath: uuidPath) as? T else { callback(nil); return }
        let first = NotificationCenter.default.addObserver(forName: Store.changedNotification, object: i, queue:
```

```

    if let item = notification.object as? T, item === i {
        if notification.userInfo?[Item.changeReasonKey] as? String == Item.removed {
            return callback(nil)
        } else {
            callback(item)
        }
    }
}
callback(i)
return [first]
}

```

The store still sends the same notifications, we're just subscribing to the notifications a different way, preprocessing the notification data and handling other boilerplate so the work in each view controller is simpler.

Looking at the massive view controller problem

Larger code sizes imply that your view controller is performing work unrelated to its primary role (observing the model, presenting views, providing them with data and receiving their actions); or that it should be broken into multiple controllers that each manage a smaller section of the view hierarchy; or your interfaces and abstractions failing to encapsulate the complexity of the tasks in your program and the view controller is cleaning up the mess.

In many cases, the best way to approach to problem is to proactively move as much functionality into the model layer as possible. Sorting, fetching or processing of data are common functions that end up in the controller because they're not part of the persistent state of the application but they still relate to your application's data and domain logic and are better placed in the model.

The largest view controller in the Recordings app is the `FolderViewController.m`. It is around 130 lines of code, making it far too small to show any significant problems.

Instead, we'll take a brief look at some large view controllers from popular iOS projects on GitHub. Please note, we are not trying to disparage these projects in any way. We are simply using the open source nature of these projects to look at why a view controller might grow to thousands of lines of code.

The line-count numbers we'll discuss are generated by *cloc* and ignore whitespace and comments.

Wikipedia's PlacesViewController

The version of the file reviewed for this book was version 2c1725a of: <https://github.com/wikimedia/wikipedia-ios/blob/develop/Wikipedia/Code/PlacesViewController.swift>

This file is 2326 lines long. The view controller includes the following roles:

1. Configuring and managing the map view which displays results (400 lines)
2. Getting user location with the location manager (100 lines)
3. Performing searches and gathering results (400 lines)
4. Grouping results for display in the visible area of the map (250 lines)
5. Populating, managing the search suggestion table view (500 lines)
6. Handling layout of overlays like the suggestion table view (300 lines)

This shows the classic trifecta of massive view controller causes:

1. More than one major view being managed (map view and suggestion table view)
2. Asynchronous tasks (getting user location) whose data is consumed but which is otherwise unrelated to the view controller
3. Model/domain logic (searching and processing results) performed in the controller layer

You can simplify the view requirements in this scene by separating the major views into their own, smaller controllers. They don't even need to be instances of `UIViewController` – they could just be child objects owned by the scene. The only work left in the parent view controller might then be integration and layout.

You can create utility classes to perform asynchronous tasks like getting user location information and the only code required in the controller is the construction of the task and the callback closure.

From an application design perspective, the biggest problem is the model/domain logic in the controller layer. This code lacks an actual model to perform searches and gather search results. Yes, there is a `dataStore` object used to hold this information but it has no abstraction around it so it is not helpful on its own. All of the actual search and data processing is performed in the controller layer by the view controller. Data fetching and processing should be handled in another location. Even work like the grouping of results for the visible area could be performed by the model or a transformation object between the model and the view controller.

WordPress' AztecPostViewController

The version of the file reviewed for this book was version 6dcf436 of: <https://github.com/wordpress-mobile/WordPress-iOS/blob/develop/WordPress/Classes/ViewRelated/Aztec/ViewControllers/AztecPostViewController>

This file is 2703 lines long. The view controller includes the following roles:

1. Constructing subviews in code (300 lines)
2. Autolayout constraints and title placement (200 lines)
3. Managing the article publishing process (100 lines)
4. Setting up child view controllers and handling their results (600 lines)
5. Coordinating, observing and managing input to the text view (300 lines)
6. Displaying alerts and the alert contents (200 lines)
7. Tracking media uploads (600 lines)

Media uploads is a domain service that could easily be moved into its own model layer service.

But 75% of the remaining code could be eliminated by improving the interfaces to other components in the program.

None of the models, services or child view controllers that this `AztecPostViewController` deals with can be used in a single line. Displaying an alert takes a half dozen lines in multiple locations for each alert. Running a child view controller takes twenty lines of setup and twenty lines of handling after completion. Even though the text view is the

custom `Aztec.TextView`, there are hundreds of lines in the view controller tweaking its behavior.

These are all examples where the view controller is being used to patch the behavior of other components that are failing to complete their own job. These behaviors should all be baked into the components where possible. When the behavior of these components can't be changed, wrappers around components should be used to adapt them. It should not be the view controller's responsibility to fix the mess left by every other component in the program.

Firefox's `BrowserViewController`

The version of the file reviewed for this book was version 98ec57c of: <https://github.com/mozilla-mobile/firefox-ios/blob/master/Client/Frontend/Browser/BrowserViewController.swift>

This file is 2209 lines long. The view controller includes over 1000 lines of delegate implementations including: `URLBarDelegate`, `TabToolbarDelegate`, `TabDelegate`, `HomePanelViewControllerDelegate`, `SearchViewControllerDelegate`, `TabManagerDelegate`, `ReaderModeDelegate`, `ReaderModeStyleViewControllerDelegate`, `IntroViewControllerDelegate`, `ContextMenuHelperDelegate`, `KeyboardHelperDelegate` and more.

What are these and what do they do?

Basically, the `BrowserViewController` is the top-level view controller in the program and these delegate implementations represent lower-level view controllers using the `BrowserViewController` to relay actions around the program.

Yes, it is the controller layer's responsibility to relay actions around the program – so this isn't a case of model/domain responsibilities spilling out into the wrong layer – but many of these delegate actions are unrelated to the view managed by the `BrowserViewController` (they merely want to access other components or state stored on the `BrowserViewController`).

These delegate callbacks could all be relocated onto a coordinator, or other abstract (non-view) controller, dedicated to handling relays within the controller layer; instead of

throwing this responsibility onto a view controller that already has plenty of other responsibilities.

Code Instead of Storyboards

Rather than using Storyboards, you can choose to define your view hierarchy in code. This change gives you more control over the construction phase, and one of the biggest advantages is that you then have better control over dependencies.

For example, when using Storyboards, there is no way to guarantee that all necessary properties for a view controller are set in `prepare(for:sender:)`. Recall that we've used two different techniques for passing model objects: a default value (as in the folder view controller) or an optional type (as in the play view controller). Neither of these techniques guarantees that the object is passed: if you forget to do so, they silently continue with either the wrong value or an empty value.

When we move away from Storyboards, we can gain more control over the construction process, and we can enforce that the necessary parameters are passed in. Note that it is not necessary to completely dispose of Storyboards. We could start by removing all segues and performing them manually instead. To construct a folder view controller, we can add a static method:

```
extension FolderViewController {
    static func instantiate(_ folder: Folder) -> FolderViewController {
        let sb = UIStoryboard(name: "Main", bundle: nil)
        let vc = sb.instantiateViewController(withIdentifier: "folderController") as! FolderViewController
        vc.folder = folder
        return vc
    }
}
```

When we create a folder view controller using the `instantiate` method, the compiler helps us and tells us we need to provide the `folder`; it is not possible to forget to provide it. Ideally, `instantiate` would be an initializer, but that's only possible if we move away from Storyboards completely.

We can apply the same technique to view construction as well. We can add convenience initializers to view classes, or write functions to construct specific view hierarchies. In

general, eschewing Storyboards allows us to use all language features in Swift: generics (e.g. to configure a generic view controller), first-class functions (e.g. to style a view or set callbacks), enums with associated values (e.g. to model exclusive states), and so on. At the time of writing, Storyboards don't support these features.

Reusing Code with Extensions

To share code between view controllers, a common solution is to create a shared superclass containing the shared functionality. A view controller then gains that functionality by subclassing. This technique can work well but has a potential downside: you can only pick a single superclass for your new class — for example, it's not possible to inherit from both `UIPageViewController` and `UITableViewController`. This technique also often leads to what we call the *god view controller*: a shared superclass that contains all shared functionality in the project. Such a class often becomes so complex that it decreases maintainability.

Another way to share code between view controllers is using extensions. If you find yourself writing a similar method in multiple view controllers, you can sometimes add it as an extension on `UIViewController` instead. That way, all view controllers gain that method. For example, we have added a convenience method to `UIViewController` that displays modal text alerts.

For an extension to be useful, it's often necessary that the view controller has some specific capabilities. For example, the extension might require that a view controller has an activity indicator present, or that a view controller has certain methods available. We can capture these capabilities in a protocol. As an example, we can share keyboard handling code, resizing a view when the keyboard shows or hides. If we use Auto Layout, we can specify that we expect the capability of having a resizable bottom constraint:

```
protocol ResizableContentView {  
    var resizableConstraint: NSLayoutConstraint { get }  
}
```

Then we can add an extension to every `UIViewController` that implements this protocol:

```
extension ResizableContentView where Self: UIViewController {  
    func addKeyboardObservers() {  
        // ...  
    }  
}
```

```
}  
}
```

We can use this technique to share code between view controllers without using subclassing.

Reusing Code with Child View Controllers

Child view controllers are another option to share code between view controllers. For example, if we wanted to show a small player at the bottom of the folder view controller, we can add a child view controller to the folder view controller, so that the player logic is contained and doesn't clutter up the folder view controller. This is easier and more maintainable than duplicating the code within the folder view controller.

UIKit has a number of built-in view controllers that add child view controllers, mostly dealing with navigation patterns: `UISplitViewController`, `UINavigationController`, `UIPageViewController`, and so on.

When you have a single view controller with two distinct states, you could also separate it into two view controllers (one for each state) and use a container view controller to switch between the two child view controllers. For example, we could split up the play view controller into two separate view controllers: one that shows the "No Recording Selected" text, and another for displaying a recording. A container view controller can then switch between the two depending on the state. There are two advantages to this approach: firstly, the empty view controller could be reused if we make the title (and other properties) configurable. Second, the play view controller won't have to deal with the case where the recording is nil: we only create and display it when we have a recording.

Extracting Objects

Most large view controllers have many roles and responsibilities. Often, a role or responsibility can be extracted into a separate object, although it's not always easy to see that. We find it helpful to distinguish between coordinating controllers and mediating controllers ([as defined by Apple](#)). A coordinating controller is application-specific and generally not reusable (for example, almost all view controllers).

A mediating controller is a reusable controller object, which is configured for a specific task. For example, the AppKit framework provides classes like NSArrayController or NSTreeController. On iOS, we can build similar components. Often, a protocol conformance (such as the folder view controller conforming to UITableViewDataSource) is a good candidate for a mediating controller. Pulling out these “conformances” into separate objects can be an effective way to make a view controller smaller. As a first step, in the folder view controller, we can extract the table view’s data source without making modifications to the implementation of the methods:

```
class FolderViewDataSource: NSObject, UITableViewDataSource {
    var folder: Folder

    init(_ folder: Folder) {
        self.folder = folder
    }

    func numberOfSections(in tableView: UITableView) -> Int {
        return 1
    }

    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        return folder.contents.count
    }

    func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        let item = folder.contents[indexPath.row]
        let identifier = item is Recording ? "RecordingCell" : "FolderCell"
        let cell = tableView.dequeueReusableCell(withIdentifier: identifier, for: indexPath)
        cell.textLabel!.text = "\\((item is Recording) ? □" : □")" \\(item.name)"
        return cell
    }

    func tableView(_ tableView: UITableView, commit editingStyle: UITableViewCellEditingStyle, forRowAt indexPath: IndexPath) {
        folder.remove(folder.contents[indexPath.row])
    }

    func tableView(_ tableView: UITableView, canEditRowAt indexPath: IndexPath) -> Bool {
        return true
    }
}
```

```
}
```

Then, in the folder view controller itself, we set the table view's data source to our new data source:

```
lazy var dataSource = FolderViewDataSource(folder)
```

```
override func viewDidLoad() {  
    super.viewDidLoad()  
    tableView.dataSource = dataSource  
    // ...  
}
```

We also need to take care of observing the view controller's folder, and change the data source's folder whenever it changes. At the price of some extra communication, we have separated the view controller into two components. In this case, the separation doesn't have too much overhead, but when the two components are tightly coupled and need a lot of communication, or share a lot of state, the overhead might be so big that it only makes things more complicated.

If you have multiple similar objects, you might be able to generalize them. For example, in the FolderViewDataSource above, we could change the stored property from Folder to [Item] (an Item is either a folder or a recording). And as a logical next step, we can make the data source generic over the Element type, and move the Item-specific logic out. The interface would look like this:

```
class ArrayDataSource<Element>: NSObject, UITableViewDataSource {  
    init(_ contents: [Element],  
         identifier: @escaping (Element) -> String,  
         remove: @escaping (_ at: Int) -> (),  
         configure: @escaping (Element, UITableViewCell) -> ()) {  
        // ...  
    }  
}
```

To configure it for our folders and recordings, we need the following code:

```
ArrayDataSource(folder.contents,  
                identifier: { $0 is Recording ? "RecordingCell" : "FolderCell" },
```

```

remove: { [weak self] index in
    guard let folder = self?.folder else { return }
    folder.remove(folder.contents[index])
},
configure: { item, cell in
    cell.textLabel!.text = "\\((item is Recording) ? " : " " \\(item.name)"
})

```

In our small example app, we don't gain much by making our code so generic. However, in larger apps, the technique can help to reduce duplicated code and allow type-safe reuse.

Simplifying View Configuration Code

If your view controller is constructing and updating a lot of views, it can be helpful to pull this view configuration code out. Especially in the case of “set and forget”, where there isn't two-way communication, this can simplify your view controller. For example, when you have a complicated `tableView(_:cellForRowAtIndexPath:)`, you can move some of that code out of the view controller:

```

override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
    let item = folder.contents[indexPath.row]
    let cell = tableView.dequeueReusableCell(withIdentifier: identifier, for: indexPath)
    cell.configure(for: item) // Extracted
    return cell
}

```

You can then move all that logic onto `UITableViewCell` or a subclass:

```

extension UITableViewCell {
    func configure(for item: Item) {
        textLabel!.text = "\\((item is Recording) ? " : " " \\(item.name)"
    }
}

```

In our case, extracting cell configuration made the view controller marginally simpler, because the original code was just one line. However, if you have cell configuration code that spans multiple lines, it might be worth it. You can also use this pattern to share

configuration and layout code between different view controllers. As an added benefit, it's easy to see that `configure(for:)` doesn't depend on any of the state in the view controller: all state is passed in through parameters and therefore it's easy to test.

Conclusion

Model-View-Controller is the simplest and most commonly used of the patterns we will discuss in this book. The other application design patterns in this book all represent – to varying degrees – a break from convention. Unless you know that you want to choose a less conventional path for your project, you should probably start your projects in MVC.

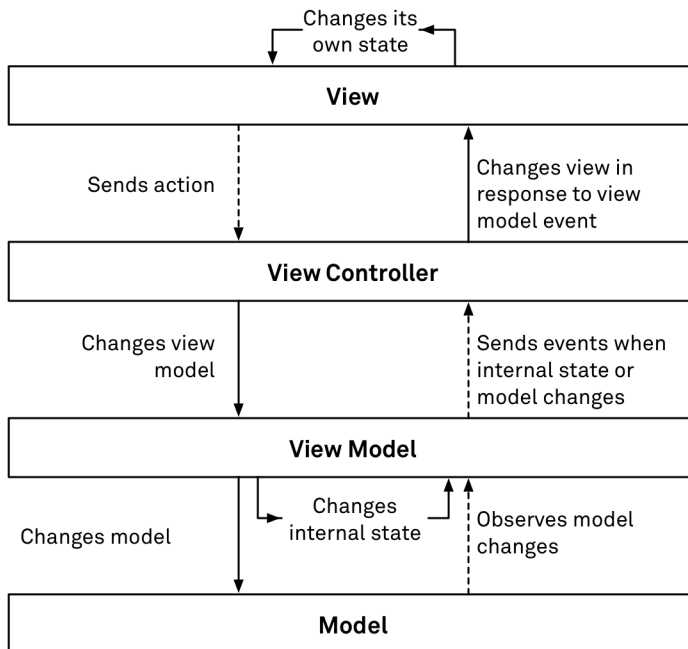
Model-View-Controller is sometimes discussed in disparaging terms by programmers advocating different architectural patterns. Claims against MVC include: view controllers aren't testable, view controllers grow too large and cumbersome, data dependencies are difficult to manage. After reading this chapter though, we hope you've seen that, while MVC has its own challenges, it is possible to write clear, concise view controllers that include full tests and cleanly manage their data dependencies.

Still, there are other ways to approach application design that alter where the challenges lie. In the following chapters, we'll show a number of alternative architectures, which each solve the problems of application design in a different way. All other patterns will include ideas that you can bring back and apply to MVC programs to solve specific problems pragmatically or produce your own hybrid patterns or programming style.

Model View ViewModel + Coordinator

4

Model-View-ViewModel (MVVM) is a pattern that aims to improve upon MVC by moving all model-related tasks (updating the model, observing it for changes, transforming model data for display, etc.) out of the controller layer and into a new layer of objects, called view-models. View-models – in common iOS implementations – sit between the model and the controller.




As with all good abstractions, there is a greater purpose than simply moving code into a new location. The purpose of the new view-model layer is two-fold:

1. To encourage structuring the relationship between the model and the view as a pipeline of transformations.
2. To provide an interface that is independent of the application framework but substantially represents the views' presentation state.

Together, these purposes address two of the biggest criticisms of MVC. The first reduces the responsibilities of view controllers by moving model-related observation and

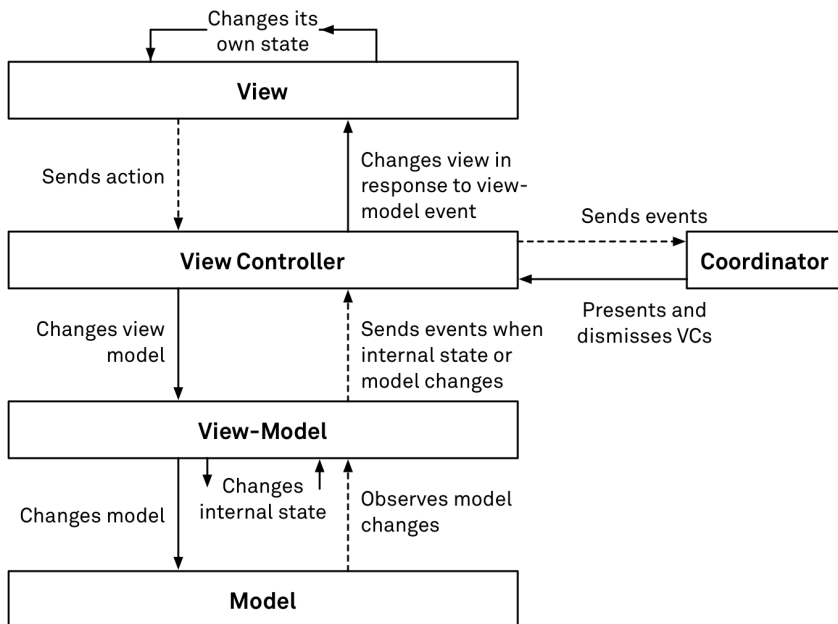
transformation tasks out of the controller layer. The second provides a clean interface to a scene's view state that can be tested independent of the application framework – as opposed to MVC's integrated testing.

To keep the view synchronized with the view-model, MVVM mandates the use of some form of bindings, i.e. any mechanism designed to keep properties on one object in-sync with properties on another object. The view controller constructs these bindings between properties exposed by the view-model and properties on views in the scene that the view-model represents.



We'll be using reactive programming to implement the bindings. This is the most common approach for MVVM on iOS. However, reactive programming is not a default inclusion in Cocoa projects and can be difficult to read if you're unfamiliar. Please read the section titled "A Quick Introduction to Reactive Programming", below, to help you with the terms and concepts.

The MVVM version of our example app additionally introduces a coordinator component. The coordinator is not a mandatory component of MVVM but it helps to relieve the view controllers of another typical responsibility: managing the presentation of other view controllers and mediating communication of model and control data between view controllers. The coordinator is in charge of managing the view controller hierarchy, so that the view controllers and view-models only need to be concerned about their particular scene. The diagram of Model-View-ViewModel + Coordinator (MVVM+C) looks like this:



In our example app the coordinator is the delegate of view controllers, and view controllers forward navigation actions directly to the coordinator. Some people prefer to forward navigation actions to the view-model first, and then have the coordinator observe the view-model for navigation events. The latter approach is useful if navigation events are dependent on the current view state or model data. Moving the navigation events from the view controller to the view-model allows for easier testing of this interplay. Our example app wouldn't have gained any practical advantage from looping navigation events through the view-model though, so we opted for the simpler solution.

Exploring the implementation

The MVVM pattern can be seen as a more elaborate version of MVC but suggesting additional components to take on some of the view controller responsibilities. Despite this similarity between the two patterns, there are parts of an MVVM implementation that may look radically different.

The primary reason for the differences is that communication between the model and the view in MVVM is not a single observer pattern interface, as in MVC, but is restructured as a data pipeline, a task which we implement using reactive programming to build the pipeline. Reactive programming a data pipeline using a series of transformation stages. This can look very different to building logic with Swift's basic control-flow statements (loops, conditions, method calls).

To limit confusion for those unfamiliar with reactive programming, we're going to summarize the important concepts in reactive programming before looking at how construction, model changes and view state changes are handled in MVVM.

A Quick Introduction to Reactive Programming

Reactive programming is a pattern for describing data flows between sources and consumers of data as a pipeline of transformations. Data flowing through the pipeline is treated as a sequence and transformed using functions with similar names to some of Swift's sequence and collection functions.

Let's look at a quick example. Imagine we had an optional string value on a model object that we wanted to observe and apply its results to a label. Assuming the string value is declared dynamic, we can observe it in a view controller's `viewDidLoad` method using key-value observing:

```
var modelObject: ModelObject!
var obs: NSKeyValueObservation? = nil

override func viewDidLoad() {
    super.viewDidLoad()

    obs = modelObject.observe(\ModelObject.value) { [unowned self] obj, change in
        if case let value?? = change.newValue {
            self.textLabel.string = "Selected value is: \(value)"
        } else {
            self.textLabel.string = "No value selected"
        }
    }
}
```

Other than the need to unwrap two layers of optional around `change.newValue`, it's fairly standard Cocoa programming; we observe the value and when it changes, update the text label.

Using RxSwift for reactive programming, we might instead write:

```
var modelObject: ModelObject!
var disposeBag = DisposeBag()

override func viewDidLoad() {
    super.viewDidLoad()

    modelObject.valueObservable.map { possibleValue -> String in
        if let value = possibleValue {
            return "Selected value is: \(value)"
        } else {
            return "No value selected"
        }
    }.bind(to: self.textLabel.rx.text).disposed(by: disposeBag)
}
```

The differences might initially seem aesthetic:

- Instead of observing the value property directly, we read a property named `valueObservable`.
- Instead of performing all the work in a single callback, we perform the purely data-related transformation in the middle – on its own – using a map transformation.
- The actual setting of the `textLabel.text` is performed automatically at the end of the chain inside the `bind(to:)` to which we pass the `rx` extension of the label's text property.

Why is this important?

Instead of having multiple places where the `textLabel.text` value is set, the text label is referenced just once, at the end. Reactive programming asks you to start at the destination – the *subscriber* of the data – and walk your way backwards through the data transformations back to the original data dependencies – the *observables*. In doing this,

it keeps the three parts of a data pipeline – the observable, the transformations and the subscribers – separate. The end observer is made simpler (handled entirely by the framework, in this case), and the source observable can include some carefully managed handling inside the model abstraction.

The transformations are both the biggest benefit of reactive programming and the point where the learning curve gets the steepest. You have probably used `map` with Swift sequences or optionals and the RxSwift version works in a similar way. Other functions in RxSwift that have Swift Sequence equivalents include `filter` (same in both), `concat` (similar to `append(contentsOf:)`), `skip` and `skipWhile` (similar to `dropFirst` and `dropWhile`), `take` and `takeWhile` (similar to `prefix` and `prefixWhile`).

It might also be worth mentioning the `flatMapLatest` transformation. This transformation observes its source and every time the source emits a value, uses that value to construct, start or select a new observable. The values emitted by the new observable are emitted from the `flatMapLatest` result. It might seem confusing but it lets you subscribe to a second observable based on the state emitted through a first observable.

It's also worth mentioning some of the types in RxSwift:

- `Observable` is a property which you can transform, subscribe or bind.
- `PublishSubject` is an `Observable` but you can also *send* values to it that will be emitted to observers.
- `ReplaySubject` is like `PublishSubject` except you can start sending before any observers have connected and new observers will receive any previously sent values cached in the “replay” buffer.
- `Variable` is a wrapper around a settable value and offers an `Observable` property so you can observe the value each time it is set.
- `Disposable` and `DisposeBag` are used to maintain the lifetime of one subscriber or multiple subscribers, respectively. When the disposable is released or explicitly disposed, the subscription is ended.

That's plenty of information to absorb. Perhaps we should move on and look at MVVM.

Construction

MVVM's approach to construction follows a similar pattern to MVC: the controller layer has full knowledge of the structure of the program and uses this knowledge to construct and connect all components.

Relative to MVC, there are three key differences:

1. The view-model must be constructed.
2. Bindings between the view-model and views must be established.
3. References to the model must be set on the view-model (not the controller).

Looking at the first of these points in the Recordings app, we've chosen to default construct a corresponding view-model as a member of each view controller:

```
class FolderViewController: UITableViewController {  
    let viewModel = FolderViewModel()  
    // ...  
}
```

This particular arrangement is a simple, low-friction approach when using storyboards and segues since the view-model does not need to be set on the view controller during segues or otherwise set after construction. The view-model is not configured with a reference to a model object at construction. The model object has to be set on the view-model at a later point.

In an alternative arrangement, the view-model on the view controller is declared as an initially nil optional and a fully configured view-model is set on the view controller at a later point. We avoided this approach because it didn't solve any data propagation issues in the Recordings app and added an optional to an otherwise non-optional type. If you ignore storyboards entirely and use manual view controller initialization, you can pass the view-model in as a parameter to the view controller and achieve the best of both arrangements.

Connecting views to initial data

In the MVVM+C pattern that we're using, it is the coordinator's responsibility to set the model objects on the view-models. The coordinator itself is a high-level controller object, which is constructed in the application delegate as the final step in the startup process, so it must ensure that any initially constructed view controllers are provided with data at this time.

```
@UIApplicationMain
```

```
class AppDelegate: UIResponder, UIApplicationDelegate, UISplitViewControllerDelegate {  
    var app: Coordinator? = nil
```

```
  
    func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions: [UIApplication.LaunchOptionsKey: Any]?) throws -> Bool {  
        let splitViewController = window!.rootViewController as! UISplitViewController  
        splitViewController.delegate = self  
        splitViewController.preferredDisplayMode = .allVisible  
        app = Coordinator(splitViewController)  
        return true  
    }  
    // ...  
}
```

In its own constructor, the coordinator ensures that the top-level view-models are provided with initial data:

```
final class Coordinator {  
    init(_ splitView: UISplitViewController) {  
        // ...  
        let folderVC = folderNavigationController.viewControllers.first as! FolderViewController  
        folderVC.delegate = self  
        folderVC.viewModel.folder.value = Store.shared.rootFolder  
        folderVC.navigationItem.leftItemsSupplementBackButton = true  
        folderVC.navigationItem.leftBarButtonItem = folderVC.editButtonItem  
        // ...  
    }  
    // ...  
}
```

The line `folderVC.viewModel.folder.value = Store.shared.rootFolder` is the most important: this is where the root `FolderViewModel` receives its data. It's also the first line of code using the RxSwift framework: the `folder` property on the folder view-model is an RxSwift Variable:

```
class FolderViewModel {
    let folder = Variable<Folder>(Store.shared.rootFolder)
    // ...
}
```

Assigning to a `Variable` has two effects: the first is that the folder view-model's `folder.value` property will return the value we have set. The second is that any observers of this variable will also receive the new value.

To explore how the views ultimately receive this data, we need to look at the implementation of the folder view-model. To observe its `folder` variable, the view-model constructs a dependent `folderUntilDeleted` observable on initialization:

```
// FolderViewModel
private let folderUntilDeleted: Observable<Folder?>

init() {
    folderUntilDeleted = folder.asObservable()
    // Every time the folder changes
    .flatMapLatest { currentFolder in
        // Start by emitting the initial value
        Observable.just(currentFolder)
        // Re-emit the folder every time a non-delete change occurs
        .concat(currentFolder.changeObservable.map { _ in currentFolder })
        // Stop when a delete occurs
        .takeUntil(currentFolder.deletedObservable)
        // After a delete, set the current folder back to `nil`
        .concat(Observable.just(nil))
    }.share(replay: 1)
}
```

We've commented each line so if you're not familiar with reactive programming, please read the comments for a step-by-step summary of this code. Reactive programming builds logic from sequences of its own transformations. These transformations encode

significant logic in a highly efficient syntax but needing to learn the common reactive programming transformations adds a steep learning curve to reading reactive programming code.

The gist of the code above is: we combine the folder observable with other model-derived data observations that might affect the logic of our view. The result is a `folderUntilDeleted` observable that will update correctly when the underlying folder object is mutated and will set itself to `nil` if the underlying folder object is deleted from the store.

The `folderUntilDeleted` observable is not observed directly by the view controller. The purpose of the view-model is to prepare all data into the correct format so it is ready to be directly bound to the view. In this case, that means extracting the required text or other properties from the folder object emitted by the `folderUntilDeleted` observable.

For example, the title displayed in the navigation bar at the top of the `FolderViewController` is prepared in the `FolderViewModel` as follows:

```
// FolderViewModel
var navigationTitle: Observable<String> {
    return folderUntilDeleted.map { folder in
        guard let f = folder else { return "" }
        return f.parent == nil ? .recordings : f.name
    }
}
```

The observable output of this `navigationTitle` property is bound to the navigation item's title by the following line in the `FolderViewController`'s `viewDidLoad` method:

```
// FolderViewController
override func viewDidLoad() {
    super.viewDidLoad()
    viewModel.navigationTitle.bind(to: navigationItem.rx.title).disposed(by: disposeBag)
    // ...
}
```

Thus we have the full pipeline of data:

1. The coordinator sets the initial model objects on the view-model of each view controller.
2. The view-model combines the set value with other model data and observations.
3. The view-model prepares the data into the precise format required for the view.
4. The view controller uses `bind(to:)` to connect these prepared values to each view.

Applying state restoration data

MVVM largely follows the same state restoration approach as Model-View-Controller, except that the data stored for each view controller comes from the view-model instead of properties stored on the view controller itself. Therefore the implementations of `encodeRestorableState` and `decodeRestorableState` on the folder view controller are almost identical to the MVC versions:

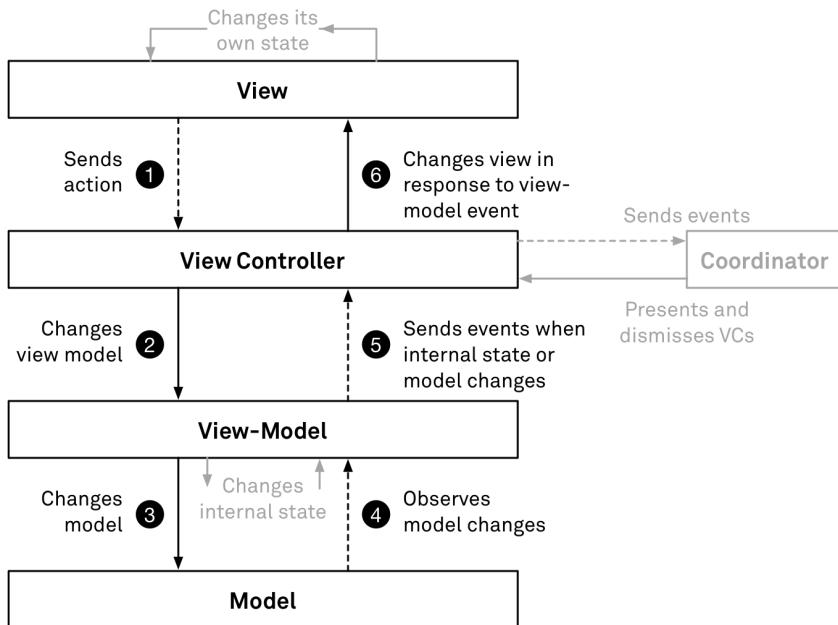
```
// FolderViewController
override func encodeRestorableState(with coder: NSCoder) {
    super.encodeRestorableState(with: coder)
    coder.encode(viewModel.folder.value.uuidPath, forKey: .uuidPathKey)
}

override func decodeRestorableState(with coder: NSCoder) {
    super.decodeRestorableState(with: coder)
    if let uuidPath = coder.decodeObject(forKey: .uuidPathKey) as? [UUID], let folder = Store.shared.item(
        self.viewModel.folder.value = folder
    ) else {
        if var controllers = navigationController?.viewControllers, let index = controllers.index(where: { $0 =
            controllers.remove(at: index)
            navigationController?.viewControllers = controllers
        })
    }
}
```

The only difference is that the code uses `viewModel.folder.value` instead of simply `self.folder`, but it's otherwise identical to the Model-View-Controller code.

Changing the model

The event loop from view to model and back in MVVM+C follows a similar path through the layers compared to MVC, with the addition of the mediating view-model inserted between the view controller and the model:



Again though, as with construction, there are aspects of the implementation style that are heavily shaped by the use of reactive programming. Below we'll look at the implementation of the steps involved to propagate the deletion of a folder from the table view to the model and back up to the view.

Step 1: Table View Sends Action

In our MVVM implementation the table view's data source is not configured in the storyboard, because we use RxSwift's table view extensions to provide the table view with its data. Therefore, what normally would be a call to the data source's

`tableView(_:commit:forRowAt:)` method is encapsulated as an event to the table view's `modelDeleted` observable.

Step 2: View Controller Changes View-Model

In MVVM, the view controller doesn't call into the model layer itself. Instead, it outsources this work to its view-model. As mentioned in the first step, we get notified about table view deletions via Rx's `modelDeleted` observable on the table view. We subscribe to this observable and call the view-model's `deleteItem` method:

```
// FolderViewController
override func viewDidLoad() {
    super.viewDidLoad()
    // ...
    tableView.rx.modelDeleted(Item.self)
        .subscribe(onNext: { [unowned self] in self.viewModel.deleteItem($0) }).disposed(by: disposeBag)
}
```

Step 3: View-Model Changes Model

Inside the view-model, the change is directly propagated to the model layer:

```
// FolderViewModel
func deleteItem(_ item: Item) {
    folder.value.remove(item)
}
```

The `remove` method on the `Folder` class removes the specified item from its children and calls `save` on the store to persist the changes:

```
// Folder
func remove(_ item: Item) {
    guard let index = contents.index(where: { $0 == item }) else { return }
    contents.remove(at: index)
    item.deleted()
    store?.save(item, userInfo: [
        Item.changeReasonKey: Item.removed,
```

```

        Item oldValueKey: index, Item.parentFolderKey: self
    })
}

```

Saving the changes in turn triggers a model-notification to be sent, which we'll use in the next step.

Step 4: View-Model Observes Model Changes:

As we've outlined in the construction section above, the view-model observes its folder variable for changes. Whenever the folder's contents change, the view-model needs to update its representation of these contents that the table view binds to. This is done in the folderContents observable:

```

var folderContents: Observable<[AnimatableSectionModel<Int, Item>]> {
    return folderUntilDeleted.map { folder in
        guard let f = folder else { return [AnimatableSectionModel(model: 0, items: [])] }
        return [AnimatableSectionModel(model: 0, items: f.contents)]
    }
}

```

folderContents emits the current contents of the folder in a way that can be bound to the table view, as we'll see in the next step.

Step 5 & 6: View Controller Observes View Model and Updates the Views

In viewDidLoad, the view controller binds the view model's folderContents property to the table view's data source using RxSwift's table view extensions (part of the [RxDataSources](#) library):

```

override func viewDidLoad() {
    // ...
    viewModel.folderContents.bind(to: tableView.rx.items(dataSource: dataSource)).disposed(by: disposeBag)
}

```

This is all we have to do in order to keep the table view in sync with the underlying data. Under the hood Rx's table view extensions take care of all the table view calls like `insertRows` and `deleteRows`.

Changing The View-State

In MVVM+C, part of the view-state — transient state of the user interface — is implicit in the view and view controller hierarchy, while other parts are explicitly represented by view-models.

In theory, a view-model is supposed to be a representation of the full state of the view within a particular scene. In common practice though, a view-model only represents the view state affected by view-actions. For example, while the current scroll position is clearly view-state, it's usually not represented by the view-model: it's neither dependent on any view-model properties, nor do other parts of the view-model depend on it. This makes the overall representation of view-state in the view-model equivalent to the view-state that would typically be held by properties on a view controller subclass in MVC.

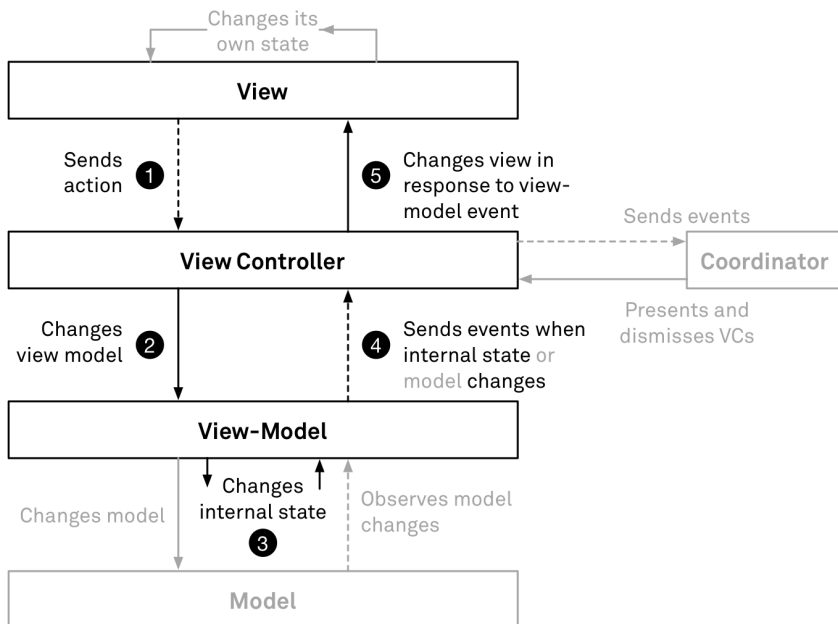
Below we'll look at the implementation details for two distinct examples of changing the view-state:

1. Changing the title of the play button in response to tapping it.
2. Pushing a folder view controller in response to selecting a sub-folder.

In the first example we only change a property of an existing view (the button's title), whereas in the second example we need to modify the view and view controller hierarchy.

Example 1: Updating the Play Button

If a view-state change doesn't affect the view controller hierarchy, the coordinator isn't involved. The view controller that receives the user action forwards it to the view-model, and reactive bindings propagate necessary view updates from the view-model to the views. This is the path taken in the first example of updating the play button's title.



Step 1: The Button Triggers an Action on the Play View Controller

The first step works the same as in the MVC variant of the example app: the play button is connected via Interface Builder to the play method on the play view controller, so play is called when the user taps the play button.

Step 2: The Play View Controller Changes the View Model

The play method makes only a single call. It sends a new event into the view model's togglePlay property:

```
@IBAction func play() {
    viewModel.togglePlay.onNext()
}
```

Step 3: The Play View Model Changes its Internal State

The call to `onNext` on the view-model's `togglePlay` property changes the audio player's state from playing to paused or vice versa. Whenever the state of the audio player changes, a new value is sent into the view-model's internal `playState` observable. From this observable we derive the `playButtonTitle` observable which maps the possible states to their respective button titles:

```
// PlayViewModel
var playButtonTitle: Observable<String> {
    return playState.map { s in
        switch s?.activity {
            case .playing?: return .pause
            case .paused?: return .resume
            default: return .play
        }
    }
}
```

Step 4 & 5: The Play View Controller Observes the View-Model and Updates the View

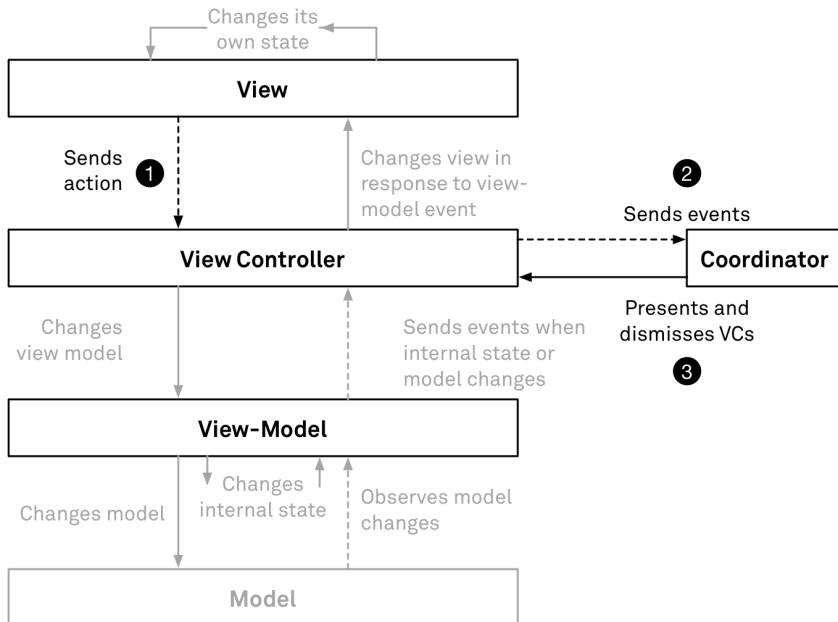
In the play view controller's `viewDidLoad` we set up a reactive binding from the view-model's `playButtonTitle` observable to the play button's title:

```
override func viewDidLoad() {
    super.viewDidLoad()
    // ...
    viewModel.playButtonTitle.bind(to: playButton.rx.title(for: .normal)).disposed(by: disposeBag)
}
```

This reactive binding observes the view-model's play button title, and propagates the changes to the actual button view.

Example 2: Pushing a Folder View Controller

When we change the view controller hierarchy — like in this example — the coordinator comes into play. The view controller receiving the user action delegates the work of updating the view controller hierarchy to the coordinator.



Step 1 & 2: The Folder View Controller Subscribes to the Table View Selection and Calls the Coordinator

In the folder view controller's `viewDidLoad` we subscribe to the table view's selection using RxSwift's table view extensions. When we receive a selection event we forward it to the folder view controller's delegate (which is the coordinator):

```
// FolderViewController
override func viewDidLoad() {
    super.viewDidLoad()
```

```
// ...
tableView.rx.modelSelected(Item.self)
    .subscribe(onNext: { [unowned self] in self.delegate?.didSelect($0) })
    .disposed(by: disposeBag)
}
```

Step 3: The Coordinator Pushes the New Folder View Controller

The coordinator's `didSelect` method checks whether the selected item is a folder. If so, it instantiates a new folder view controller from the storyboard and pushes it onto the navigation stack:

```
extension Coordinator: FolderViewControllerDelegate {
    func didSelect(_ item: Item) {
        switch item {
        case let folder as Folder:
            let folderVC = storyboard.instantiateViewController(with: folder, delegate: self)
            folderNavigationController.pushViewController(folderVC, animated: true)
            // ...
        }
    }
}
```

Similar to MVC, the `UIView/UIViewController` hierarchy still implicitly represents the view state. However, two tasks that view controllers usually handle are factored out: the view model transforms model values into the data needed by the views, and the coordinator manages the flow of view controllers.

Testing

Not yet published.

Discussion

At first glance, MVVM+C looks like a more complex pattern than Cocoa MVC; the view-model adds another layer you have to manage. At the implementation level

though, the code can become simpler if you stick to the pattern consistently. Alas, simpler doesn't necessarily mean easier to write; until you're familiar with common transformations, reactive code can be difficult to write and frustrating to debug. On the bright side, the carefully structured data pipelines are often less error prone, and simpler to maintain in the long run.

By moving model observing and other presentation and interaction logic into an isolated class structured around data-flow, MVVM addresses the problems associated with unregulated interaction of state in MVC view controllers. Since this is the most significant problem exacerbated when Cocoa view controllers grow over-large, this change goes a long way towards mitigating the massive-view-controller problem in MVC. There are other ways that view controllers (and view-models) can grow large so refactoring is still necessary on an ongoing basis.

Another problem commonly encountered in MVC — model and view getting out of sync because the observer pattern isn't applied strictly — is addressed by reactive bindings that bind view-model properties to view properties. This pattern unifies the code paths responsible for the initial configuration of views and their subsequent updates, thus making it less prone to error.

The introduction of reactive programming isn't without its drawbacks though: it is one of the biggest obstacles to understanding and writing an MVVM app like the Recordings example application. Reactive programming frameworks come with a steep learning curve and it takes a while to adjust to this style of programming. While the idea of reactive programming is conceptually elegant, reactive programming frameworks rely on highly abstract transformations and large numbers of types, misuse of which can leave your code unparseable by humans.

MVVM with less Reactive Programming

Reactive programming is a technique with its own set of trade-offs, being more attractive to some and less to others. For some programmers then, the reliance of MVVM on this technique presents a disincentive to using the pattern. Therefore, in this section we will look at using the MVVM pattern with less or no use of reactive programming.

In the [original MVVM+C implementation](#) we used reactive programming within the view-model and as a binding between the view-model's interface and the views. We'll explore two variants: first we look at a view-model without any internal reactive

programming, and then we look at an alternative to reactive bindings for updating the views when the view-model changes.

Implementing a View-Model Without Reactive Programming

For this variant, we will refactor the folder view-model to function without any internal use of reactive programming. We want to keep the same view-model interface that we previously had – including the exposed observables for the navigation title and the table view’s contents. This allows us to keep using reactive bindings in the view controller to bind the view-model’s outputs to the corresponding views.

In the original implementation, the exposed observables were implemented in terms of other, internal observables. For example, the navigationTitle observable is created by mapping over the folderUntilDeleted observable:

```
// FolderViewModel
var navigationTitle: Observable<String> {
    return folderUntilDeleted.map { folder in
        // ...
    }
}
```

Since we want to get rid of internal observables like folderUntilDeleted, we have to create the exposed observables differently. For the navigation title, we create a private ReplaySubject through which we can pass in new values, and expose it to the outside world as an observable:

```
// FolderViewModel
private let navigationTitleSubject = ReplaySubject<String>.create(bufferSize: 1)
var navigationTitle: Observable<String> { return navigationTitleSubject }
```

The folder contents observable, which is used to provide the table view with its data, is refactored in the same way.

The next step is to observe model layer for changes and to react to them by sending new values into the exposed observables, if the change affected the data the view-model is serving for display. We do this by observing the store’s change notification, just as in the MVC implementation of the example app:

```
// FolderViewModel
init() {
    NotificationCenter.default.addObserver(self, selector: #selector(handleChangeNotification(_:)), name: NSNotification.Name.FolderViewModelDidChange, object: nil)
}
```

The initializer sets up the observation that invokes the `handleChangeNotification` method for every change in the underlying data. Within this method we check whether the change affects the parent folder of the displayed content. If so, we have to handle two cases: either the folder was deleted, or it was changed (e.g. its title). If it's the former, we send empty values into the observables. If it's the latter, we just re-set the view-model's folder property:

```
// FolderViewModel
@objc func handleChangeNotification(_ notification: Notification) {
    if let f = notification.object as? Folder, f === folder {
        if notification.userInfo?[Item.changeReasonKey] as? String == Item.removed {
            navigationTitleSubject.onNext("")
            folderContentsSubject.onNext([AnimatableSectionModel(model: 0, items: [])])
        } else {
            folder = f
        }
    }
    // ...
}
```

If the change was not a delete, then we re-set the folder property to trigger its property observer:

```
// FolderViewModel
@objc func handleChangeNotification(_ notification: Notification) {
    // ...
    if let f = notification.userInfo?[Item.parentFolderKey] as? Folder, f === folder {
        folder = f
    }
}
```

The property observer on the folder property will send the latest values into the exposed observables:

```
// FolderViewModel
var folder: Folder = Store.shared.rootFolder {
    didSet {
        navigationTitleSubject.onNext(folder.parent == nil ? .recordings : folder.name)
        folderContentsSubject.onNext([AnimatableSectionModel(model: 0, items: folder.contents)])
    }
}
```

For a simple view-model like this, we can express the internal logic easily without using reactive pipelines. If the view-model is more complex, especially if it has to handle more internal state, reactive programming has more opportunities to shine by making the implementation more robust. The full code of the reactive-free folder view-model can be found on [GitHub](#).

Binding a View-Model to Views Without Reactive Programming

For this variant, we'll refactor the play view-model and play view controller to entirely remove reactive programming. With bindings often considered essential to MVVM, the result might be considered Model-View-Presenter (MVP).

In contrast to the folder view-model, the play view-model has many more outputs — one for each interface element of the player. For example:

```
// PlayViewModel
var progress: Observable<TimeInterval?> {
    return playState.map { $0?.currentTime }
}
var isPaused: Observable<Bool> {
    return playState.map { $0?.activity == .paused }
}
var isPlaying: Observable<Bool> {
    return playState.map { $0?.activity == .playing }
}
var nameText: Observable<String?> {
    return recordingUntilDeleted.map { $0?.name }
}
// ...
```


Since our goal is to remove the dependency on RxSwift entirely, we not only have to change the internal implementation of the view-model, but also its interface to the view controller. We could replace each observable with a delegate method, which informs the view controller about a change for a specific piece of displayed data. However, we'll use a different approach to reduce complexity: we create a single struct that contains all data needed to configure the views:

```
// PlayViewModel
struct ViewState {
    var navigationTitle = ""
    var hasRecording = false
    var timeLabelText: String? = nil
    var durationLabelText: String? = nil
    var sliderDuration: Float = 1.0
    var sliderProgress: Float = 0.0
    var progress: TimeInterval? = nil
    var isPaused = false
    var isPlaying = false
    var nameText: String? = nil
    var playButtonTitle = String.play
}
```

Whenever the underlying recording or the current playback state changes, we update the `viewState` property on the view-model with the latest values:

```
// PlayViewModel
private func updateViewState() {
    if let r = recording {
        viewState = ViewState(
            navigationTitle: r.name,
            hasRecording: true,
            timeLabelText: (audioPlayer?.state.currentTime).flatMap(timeString),
            durationLabelText: (audioPlayer?.state.duration).flatMap(timeString),
            sliderDuration: (audioPlayer?.state.duration).map(Float.init) ?? 1.0,
            sliderProgress: (audioPlayer?.state.currentTime).map(Float.init) ?? 0.0,
            progress: audioPlayer?.state.currentTime,
            isPaused: audioPlayer?.state.activity == .paused,
            isPlaying: audioPlayer?.state.activity == .playing,
            nameText: r.name,
        )
    }
}
```

```

        playButtonTitle: audioPlayer?.state.activity.displayTitle ?? .play
    )
} else {
    viewState = ViewState()
}
}

```

Finally, we use a property observer on the `viewState` property to inform the view-model's delegate (the view controller) about the change:

```

// PlayViewModel
private var viewState = ViewState() {
    didSet {
        delegate?.updateViewState(viewState)
    }
}

```

The play view controller implements a single method to update its views when the view-models view state changes:

```

extension PlayViewController: PlayViewModelDelegate {
    func updateViewState(_ state: PlayViewModel.ViewState) {
        navigationItem.title = state.navigationTitle
        activeItemElements.isHidden = !state.hasRecording
        noRecordingLabel.isHidden = state.hasRecording
        progressLabel.text = state.timeLabelText
        durationLabel.text = state.durationLabelText
        progressSlider.maximumValue = state.sliderDuration
        progressSlider.value = state.sliderProgress
        playButton.setTitle(state.playButtonTitle, for: .normal)
        nameTextField.text = state.nameText
    }
}

```

Interestingly, after removing its dependency on RxSwift entirely, the implementation of the play view-model looks very similar to that of the play view controller in the MVC version of the app. It has to accomplish exactly the same tasks, except that it doesn't interact directly with UIKit. Whenever the play view controller would update a view, the refactored play view-model just changes a property on its view state.

In this approach we still gain the isolated testability of view-models without the reliance on the application framework. However, we lose the robustness of change propagation that reactive bindings provide. The unified view state struct we've used to replace view bindings comes close to the robustness of a correctly set up reactive pipeline, but it comes at a cost: we've lost the granularity of the view updates. Any change will cause all view properties to be re-configured.

This is not a problem for the play view controller, but it would be a problem for the folder view controller, since it needs fine-grained information about the table view changes to drive animations. This could be handled by diffing the new data against the old, providing the same functionality that the RxDataSources extensions provided in the MVVM implementation, under the hood.

Lessons to be Learned

Even if your codebase doesn't follow the MVVM pattern, there are insights to be gained that can be applied to Cocoa MVC as well.

Introducing Additional Layers

MVVM offers lessons about abstractions that can be applied to any codebase: in particular, that data pipelines can be structured in terms of data flow in a series of stages from more abstract to more specific.

MVVM suggests a single type of pipeline between specific points in the program. However, it is possible to build pipelines with additional stages and covering different aspects of the program – including app-models, session-models, use-cases, flow-models and more.

An app-model is a model that might sit between the view-models and the model, handling state and modes that span between view controllers and whose state might dictate whether particular view-models are possible. A session-model tracks details about the current login and might need to sit between the view-model and all network requests. A use-case is a view into a slice of model data and associated state shared by different view-models. A flow-model is a model-like representation of navigation state – like a view-model for the coordinator.

We suggest you do not introduce additional layers prematurely. Evaluate carefully whether the change makes your code simpler: easier to understand, less prone to casual errors, and easier to maintain. There's no point to additional abstractions if they don't improve your ability to write new code.

Coordinators

Coordinators are a pattern that is independent of the MVVM architecture. It can be applied to Cocoa apps as well in order to alleviate view controllers from one of their responsibilities and to decouple them. Whenever a view controller would present another one — by pushing it onto the navigation stack, presenting it modally, etc. — it calls a method on the coordinator instead. This allows you to manage the view controller hierarchy in a central place and to design your view controllers independently of other view controllers.

A similar separation can be achieved without introducing a separate coordinator object. You can also draw a strict separation between view controllers managing the presentation of other view controllers, and view controllers managing the UI. For more details on this variation of the coordinator pattern please refer to Dave DeLong's [4-part blog series](#).

Data Transformation

Another lesson to be learned from MVVM is the benefit of pulling out data transformation logic from the view controller. One responsibility of the controller layer in Cocoa MVC is to transform the model data into the display data that's needed to configure the views. Often that simply means transforming strings, numbers, or dates on a model object into their displayable forms. Even in such simple cases pulling out this code cleans up the view controller and increases testability at the same time.

The benefits become more apparent when data transformation involves more complex logic than simple formatting operations, e.g. when your views rely on information of what has changed. You might have to compare or diff new data against old data or integrate data from several model objects for display. All these operations can be cleanly separated from the rest of the view management code.

Fetch and Subscribe

As detailed in the chapter about [Model-View-Controller](#), a common source of bugs occurs when we fail to follow the observer pattern strictly, leading to the view and model falling out of synchronization. In Cocoa MVC, the observer pattern is enforced only by your own discipline, and requires two code paths: one for configuring the views initially, and another one for subsequent updates.

When we use reactive programming to bind the view-model to the views, as well as for data transformation within the view-model, the code paths for initial view configuration and subsequent updates are unified. Once the reactive pipeline has been configured correctly, subsequent model changes will propagate to the views automatically.

We can design the code in a Cocoa MVC app in a way that at least encourages the consistent use of the observer pattern, and that makes it less likely to forget to subscribe to subsequent changes. One technique to achieve this is to have an API on your model to subscribe to model notifications, which calls you back immediately. This unifies the code paths for initial configuration and later updates.

We can go even one step further in enforcing this pattern by designing the model API such that subscribing is the only way to retrieve data. We describe this technique in more detail in the chapter about [View-State Driven Model-View-Controller](#).