はじめに

本書はiOSアプリにおける設計について解説した入門書です。MVCやMVVMなどの代表的なアーキテクチャパターンの解説にとどまらず、そもそも設計とは何か？という視点から各パターンを紹介し、そして実際の現場でどのように適用されているかを解説しています。

モバイルアプリ開発はハードウェアの性能向上や、求められる要件の多様化により、年々複雑さが増しています。そうした中での開発前の設計行為や、その設計を開発中に適宜見直すことは継続的な開発に欠かせません。特にiOSアプリ開発では特定のアーキテクチャパターンが公式に推奨されているわけでもなく、みなさんも現場で試行錯誤していることと思います。

設計はそのプロジェクトの要件やチーム規模、経験などによって左右されます。この設計パターンが絶対に当てはまる！という、いわゆる銀の弾丸はありません。ですが本書で解説しているような設計を事前に知っておくことで、弾の命中率は上げられると考えています。

本書の執筆にあたって、著者一同その設計パターンの歴史から学び、iOSアプリ開発に落とし込むとどうなるかを想定して解説しています。執筆陣も調べてはじめてわかったことが数多くあります。アーキテクチャパターンが生まれた背景を踏まえ、できるだけ平易に、そして実務で役に立つ形で提示できるように心がけました。そのため本書で紹介しているサンプルコードのみが正解ではありません。読者のみなさんが今適用しているプロジェクトとの差異を感じることもあるでしょう。差異を感じたときにサンプルコードが正解と決めつけず、なぜ違うのかから考えてみてください。正解はきっとその思考の先にあります。

本書で紹介したパターンは、執筆時点の開発現場で使われているであろうメジャーなパターンを選定していますが、今後新たなアーキテクチャパターンが登場することもあるでしょう。しかしこの本で紹介した「設計するとはどういうことか」という根本的な概念は、今後廃れることはないと考えます。むしろ新しいパターンへの理解を促進するものであることを願っています。

この本を土台にして開発チームで自分たちのプロジェクトに最適な設計を議論してみてください。この本で得た知識をベースに適切な議論がなされ、合意が得られれば、それがきっと最適な設計です。最適な設計はよりよい開発につながるだけでなく、最終的にはアプリを使うユーザーの体験にもつながることでしょう。読者のみなさんがこの本から得た設計という手法を通して、よりよい開発環境が得られることを心から望んでいます。

（著者代表 松館 大輝／@d_date）

謝辞

2018年5月から始まり、約半年を要した本書の執筆も終わりを迎え、いよいよみなさまのお手元にお届けできます。

この本を執筆した著者陣は、日頃から記事を書いたり、勉強会やカンファレンスで登壇することには慣れています。これは本の執筆をマラソンとするならば、短距離走に相当します。マラソンに不慣れな我々のよき伴走者として、最後までさまざまなご指摘・サポートをいただきました編集の加藤さん、永野さん、横田さんには頭が上がりません。特にKuniwakさんには、その言語にとらわれない幅広く深い設計・テストに対する知見から、本書がよりよいものになるようにご助言いただきました。また、アーリーアクセス・ベータ版にいただきましたコメントを、最後の仕上げとしてフィードバックできました。コメントいただいたみなさまありがとうございます。

本書はクラウドファンディングで、600人のご支援を前提に出版ができるというプロジェクトでした。ありがたいことに、募集開始をしたその日に成立でき、こうして謝辞を書くころには1375人以上のご支援が得られました。ご支援いただきましたみなさまに著者一同心より御礼申し上げます。

サンプルコード

次のリポジトリに本書のサンプルコードが掲載されています。

- https://github.com/peaks-cc/iOS_architecture_samplecode

各章でサンプルコードの指定がある場合は、こちらもあわせて確認ください。

免責事項

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた開発、製作、運用は、必ずご自身の責任と判断によって行ってください。これらの情報による開発、製作、運用の結果について、著者はいかなる責任も負いません。

第1部
設計を知る

第1章

設計するということ

関 義隆／@takasek

設計とは何でしょうか。設計はなぜ必要なのでしょうか。設計パターンのカタログに飛びつく前にまず、設計の意義を考えてみましょう。

1.1 近年のiOS開発事情

iOSというものが生まれて10年が経ちました。この10年で起こったことは、アプリの複雑化です。

アプリでできることが増えた

アプリの起動経路が多様化し[注1)]、デバイスで取得できる情報の種類も増加しました[注2)]。

端末のレイアウトも多様化しました。320×480、ピクセルとポイントが1対1で紐付いていた時代はとうに過ぎました。iPhone Xの登場以降、Safe Areaの概念も導入されました。iPadやiPhone 8 Plusでは、画面を2分割してアプリを表示することもあります。どのような機種でも破綻なくレイアウトを成立させる必要があります。

ユーザーインタラクションへの細かい反応も要求されます。WWDC 2018のセッション[注3)]では、トランジションの途中であっても中断・復帰可能なインタラクションが推奨されました。

できることが増えれば、それだけ管理する状態も増えます。アプリで起こりうる状態のパターンを洗い出し網羅的に対処できるコードになっていなければ、想定外の挙動がバグを引き起こします。

頻繁・継続的なリリース

近年のアプリは頻繁かつ継続的なリリースを要求されます。

リリースサイクルを短縮することにはさまざまなメリットがあります。競合アプリに先駆けて機能を投入できますし、ユーザーの反応をもとにきめ細かなチューニングもできます。小幅な改修をユーザーにさらすことは不具合のリスクを最小化します。

一方で、改修によってバグを作り込むことは避けなければなりません。既存の挙動を保証することをリグレッションテストといいますが、人の手によるテストは非効率ですし、ミスを生みがちで

注1） 通知、Universal Links、Background fetch、Siri Shortcuts など。詳しくは第11章をご参照ください。
注2） Location、BLE、HealthKit、NFC など。
注3） Apple Inc. (2018)『Designing Fluid Interfaces』https://developer.apple.com/videos/play/wwdc2018/803/

す。とはいえ自動テストはどんなコードに対しても行えるものではありません。テストを前提としたコードを書かなければアプリの品質は保てず、継続的なリリースに支障が生じます。

アプリの大規模化

ひとつのアプリの中で、画面数もユーザー数も増えました。

アプリ内のデータやデザインの整合性を保つ労力はその画面数に応じて増大します。アプリの規模をスケールさせるためには、それぞれの画面でコンポーネントやロジックを再利用できるようにしなければなりません。そうでなければプロジェクトのコードは似たようで微妙に違うコピペの山となり、管理不能になるでしょう。

また、ユーザーが増えればバグのインパクトも大きくなります。Crashlyticsなどのクラッシュログ収集ツール上のクラッシュフリーレートをどれだけ上げられるかでアプリの評価は大きく変わってきます。いかに迅速に不具合を見つけられるか、あるいは不具合が起こらないようなしくみを作るかが重要です。

プロジェクトの長期化

アプリを作ったなら、ユーザーには長く使われたいと思うものです。

長い歴史の中では、開発コミュニティの環境も移り変わります。利用するライブラリやフレームワークの改修への対応も起こります。永続化フレームワークをCore DataからRealmに移行したり、あるいは言語自体をObjective-CからSwiftに書き替えた経験をもつ読者もいらっしゃるでしょう。

特定のフレームワークを前提としたコードは移行が難しかったり、移行が致命的なバグを生んでしまったり、最悪の場合は移行不可能と判断されます。サポートが打ち切られセキュリティリスクを抱えるライブラリを負債として使い続けるような事態は起こしたくありません。

ライブラリはあくまで開発を支援するものです。特定のライブラリを使いたくてアプリを開発しているわけではありません。環境が変わってもその影響を最小限にとどめ、なるべく本来やりたい機能開発に集中できるような「変化を織り込んだ」構造を整備する必要があります。

チーム開発での分業

ひとつのアプリを多人数のチームで開発する状況も増えました。自分以外のメンバーがコードを書くことを考えなければなりません。

メンバーの分担の単位は、各画面単位かもしれませんし、同じ画面内の画面表示とビジネスロジックでの分担かもしれません。いずれにせよメンバー全員がプロジェクトの全貌を理解し、自分のコードの影響範囲を把握し続けることは不可能です。

もしコードに差し込まれた分岐の意図が書いた人にしか分からないとしたら。あるいは変更の影響範囲を調査するためにプロジェクト全体を検索し、触ったこともないコードを丹念に読む必要があるとしたら。まるで地雷原を歩くような気分で、開発の歩みは極端に鈍ってしまうでしょう。

綿密なコミュニケーションでカバーすべきでしょうか。しかし余計なコミュニケーションコストは開発の障害になります。しかも人数比でそのコストは増大します。

新規メンバーがプロジェクトに入ってきたときにも、迅速にキャッチアップできるようにしておきたいものです。

複数人が同じファイルを同時に編集するとコンフリクトやバグの元になります。それぞれの作業は影響のないようにしたいものですが、うまく影響範囲が切り分けられないプロジェクトではしばしば衝突事故が発生してしまいます。

各メンバーが自分の領域に集中でき、必要に応じて他所を参照したときにも意図が伝わりやすい、分業に適したコードが必要です。

1.2 「iOSアプリ」「設計」「パターン」

関心の分離[注4)]という概念があります。

すべてのコンピュータシステムはなんらかの問題への関心をもち、その問題に対する解決策を提供します。しかし問題は複雑で、いざ解決しようにもさまざまな関心事が絡みあっています。レイアウトしたい、データを組み立てたい、通信したい、デバイス情報を受け取りたい、計算したい、文字を解釈したい、状態を管理したい…。挙げればきりがありません。

複雑な問題は、より単純な問題の群として切り分けるべきです。そのためにはシステムが部品（**モジュール**）の集合として構成される必要があります。それぞれの部品は狭い関心をもち、ひとつの問題にだけ対処します。部品が正しく組み合わさって動けば、総体として大きな問題を解決できるはず。

これがソフトウェア設計の根幹にある考え方です。

かつて私たちiOSアプリ開発者は、メモリの確保や解放を自らの手で行わなければなりませんでした。インスタンスを作るたびに確実に参照カウントのインクリメント処理を行い、使い終わるときにはデクリメントするコードを書いていました。メモリリークが発生したときには目を皿のようにしてコードを追い、カウントのミスを探しました。しかし2011年に登場した自動参照カウント（ARC）という言語機能によって、その役割は言語のランタイムに隠蔽されるようになりました。

2012年にはAuto Layoutが登場しました。何台もの検証機やシミュレータをとっかえひっかえしながらレイアウト崩れを確認していた作業から、私たちは少しだけ解放されました。

AppleはiOSアプリ開発に必要な関心事を、UIKitをはじめとしたさまざまなフレームワークへと分離し提供しています。そうやって私たちは、アプリの本質的な関心事に集中できるようになってきました。

それでは私たちはそのフレームワークに乗っていれば十分でしょうか？いいえ、違います。すでに述べたように、私たちの作るアプリはもはや、フレームワークにのっとった上でもまだ複雑すぎます。

注4) Edsger W. Dijkstra (1974)『On the role of scientific thought』https://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html

複雑な問題は、より単純な問題の群として切り分けましょう。

そこで強い味方になるのが**設計パターン**です。設計パターンは、**再現性のある問題に対する共通の解決策**のことです。

開発しているアプリが解決したい問題の形はさまざまですが、長いソフトウェア開発の歴史の中でまったく新しい問題はそうありません。

過去の達人たちは、何度も設計を繰り返す中で、さまざまな問題に対処するコードが共通して同じパターンに行き着くことを発見しました。彼らはそういった解決策に名前をつけ、再利用しやすいようにカタログ化していきました。

有名なパターンには、GoFのデザインパターンと呼ばれる23種類のパターンがあります。

GoFのデザインパターンの登場は1994年と古く、適用粒度がさまざまであったり、あまり違いが明確でないものがあったりします。また、モダンなプログラミング言語では言語機能として吸収されるような素朴なパターンもあります。

そのように時代を差し引いて見る必要はありますが、普段よく目にする設計パターンはここに由来するものが多いです。頭に入れておくことはきっとあなたの設計の基礎力に役立ちます。本書でも「Mediator」「Observer」「Singleton」等の概念を知っていると理解が容易になる記述があります。

GoFのデザインパターンの他にも、さまざまなプログラミングの分野で多種多様な設計パターンが編み出されました。

Column. ドメイン駆動設計

ドメイン駆動設計（DDD）という設計思想があります。

ドメインとは、iOSアプリでいえばUIの都合や通信・永続化の処理を除いた、アプリが解決したい問題領域の知識そのものです。ドメイン知識のみをレイヤーとして切り出し、そこでは非技術者の仕様策定者とも同じ言語を使い、モデリングを反復的に深化させることが価値の高いシステムを生み出すことに繋がる、というのがDDDの思想です。

ドメインを切り出す動機や、その成立過程については第4章をご参照ください。

佐藤匡剛氏[注5)]は、書籍「エリック・エヴァンスのドメイン駆動設計」は各章の説明がパターンの記述形式にのっとっているため、これはパターン本のひとつだと断じています。本書で紹介するパターンにもDDDに由来するものがあります。

詳しく知りたい方は、書籍あるいは、DDDコミュニティが公開している軽量なサマリ[注6)]をご参照ください。2003年にEric Evans氏が原則や概念を説き[注7)]、2013年にVaughn Vernon氏がより具体的な実装方法について述べています[注8)]。

注5) 佐藤匡剛(2007)『DDD難民に捧げるDomain-Driven Designのエッセンス』https://www.ogis-ri.co.jp/otc/hiroba/technical/DDDEssence/chap1.html
注6) InfoQ.com (2009)『Domain Driven Design (ドメイン駆動設計) Quickly 日本語版』https://www.infoq.com/jp/minibooks/domain-driven-design-quickly
注7) Eric Evans (2011)『エリック・エヴァンスのドメイン駆動設計』和智右桂・牧野祐子訳、翔泳社
注8) Vaughn Vernon (2015)『実践ドメイン駆動設計』高木正弘訳、翔泳社

1.3 パターンを知るメリット

パターンを知ることのメリットはいくつかあります。

問題を定型化してとらえられる

GoFがデザインパターンの本を編纂した目的は、達人の生産性の鍵がそこにあると見抜いたからです。

問題への解決策を見つけ出すまでの道は険しいものです。しかし既知のパターンを通して問題をとらえ直せば、得体の知れなかった魔物が、慣れ親しんだ理解可能な構造に見えてくるのです。わざわざ解決策を「再発見」する必要はありません。

「この問題、ゼミで見た！」という状況です。

解決策を客観的に比較できる

ある問題をパターンを通してとらえるとき、観点によっては複数のパターンが適用できそうに見えるケースもあるでしょう。それぞれのパターンの強み・弱みも、先人によってすでに検討済みです。似ているふたつのパターンの差分を確認すれば、どちらのほうが適しているのか客観的に判断できます。

パターンによっては何らかの前提を持っていることがあります。成立するケースが限定されている、利便性のため必要なハックがあるなどです。そういった実装の先での注意点についても、すべて先人が地図を残してくれているのです。なんと心強いことでしょう。

細かなトレードオフを把握した上でパターンを使い分けることで、手戻りは圧倒的に少なくなります。脳に蓄えたパターンの数は、そのままあなたが選べる武器の数です。

メンバーの共通言語となる

設計が複雑になると、各部品の関連も一目では理解できなくなります。関連図を書いてゼロから設計思想を説明することはたいへんですし、メンバー間で認識の齟齬も生まれます。しかし、そこに「これはXXパターン」と名前が与えられていれば誰もが一瞬でその構造をイメージできます。効率よく深い設計の議論ができることは、開発の生産性に大きく貢献します。

第 2 章

設計にパターンを適用する前に

関 義隆／@takasek

第 1 章にて、設計とは**関心の分離**によって**複雑な問題を単純な問題の群として切り分けること**だと述べました。切り分けられた問題の解として、しばしば**設計パターン**が使えます。**パターン**は、特定の問題をすばやく解決するための武器となります。

しかしパターンのカタログを脳内に蓄えれば万全な設計ができるのかといえば、それは違います。武器を振るうには、まず敵の姿を正しくとらえられなければなりません。問題が適切に切り分けられていなければパターンも使えないのです。

問題はどのように切り分けるべきでしょうか。パターンはいつどのように使うものなのでしょうか。

それを語るためにはよい**設計のプロセス**を整理する必要があります。

2.1 問題領域、責務、モジュール

解決すべき問題の領域は**責務**という言葉で表現されます。コードの中では、ひとまとまりの責務の置き場として**モジュール**[注1)]が用意されています。

責務分割が適切に行われた状態は、モジュール内で責務がまとまっており（高凝集）、モジュール同士では責務が分かれている（疎結合）状態とされます。

あるモジュールの責務は、他のモジュールの責務とは分離されているはずです[注2)]。にもかかわらず、一度分離したはずの責務の境界が改修を繰り返すうちに曖昧になり破綻してしまうことは、しばしば起こります。

2.1.1 切り分けられた責務の境界を守る

Swift の Foundation を利用するときですら、私たちは間違いをおかします。

たとえば、`Date` 型の extension として「昨日の日付を取得する」処理や「`yyyy/MM/dd` というフォーマットで文字列化する」処理を生やしたことはないでしょうか。

`Date` 型のドキュメントによれば、`Date` とは「いかなるカレンダーやタイムゾーンからも独立した、ある時間軸上の一点の表現」です[注3)]。

注 1） 本章では class、struct などの型も含めてモジュールと呼びます。Swift 言語における `module` ではなく、より一般的な意味の「部品（モジュール）」だと考えてください。

注 2） もちろん「責務 A と B を取りまとめる、抽象度の高い責務」というのはありえます。

注 3） https://developer.apple.com/documentation/foundation/date

一方、Calendar型は「年月日や曜日というカレンダーの単位と、絶対時間軸上の一点（Date）との関係を定義し、Dateの計算・比較の機能を提供する」役割を有しています[注4)]。

つまり先述したようなextensionは、Dateではなく Calendarの役割のはずなのです。いくら責務の正しい型があっても、間違った理解で拡張されては元も子もありません。

設計が正しくありつづけるためには、責務の境界を理解し、改修に際して崩さないようにしなければいけません。

普段、私たちはFat View Controllerに対して「まあ、AppleがUIViewControllerなんていうViewだかControllerだか分からないような奴を用意したからね…」と生暖かい目線を向けがちです。しかし一概にAppleを責めることはできません。AppleはUIViewControllerのドキュメント[注5)]にて、その役割を次の４点であると明確に定義しています。

- データ変化に応じてViewsの内容を更新
- ユーザーインタラクションへの反応
- レイアウト管理
- 他のオブジェクトとの連携

また「View Controller Programming Guide for iOS[注6)]」にはこうも書かれています。

> View ControllerはViewからの入力を検証し、Data Objectに必要なフォーマットにパッケージすることもできます。しかし、実際のデータを管理する役割は最小限にしなければなりません。

すべての開発者がドキュメントをしっかり読み、定義された以外の役割をView Controllerから排除していれば、明確な責務のままでいられたはずなのです。

そんなのは無茶だと思われたでしょうか。筆者もそう思います。ドキュメントだけを頼りに責務を正確に把握し続けるのは、無茶なことです。しかし責務の境界は守らなければなりません。

何よりも重要なのは、責務を明確にイメージできる**名前づけ**です。「ViewController」という名前では、Fatになるのは半ば必然です[注7)]。同じように〜Manager、〜Service、〜Utilといったクラス名も、責務がイメージできず肥大化や侵食に繋がるアンチパターンといえます。あるいは、本来は明確な名前をつけられる複数のモジュールをひとつにまとめてしまっているサインと考えてもよいでしょう。あるUtilがConverterとValidatorとFormatterとSenderを兼任していたら、それらは分けるべきである可能性が高いです。

これはモジュールに限ったことではありません。メソッドや変数においても、責務をイメージできない名前をつけた途端に設計は腐敗への道を辿ります。

注 4） https://developer.apple.com/documentation/foundation/calendar
注 5） https://developer.apple.com/documentation/uikit/uiviewcontroller
注 6） Apple Inc.（2015）『View Controller Programming Guide for iOS』https://developer.apple.com/library/archive/featuredarticles/ViewControllerPGforiPhoneOS/index.html
注 7） 念のため、「UIViewControllerのサブクラスに〜ViewControllerと名付けるのを避けよう」という主張ではありません。

2.1.2　正しい名前に向き合う

「アプリ利用ユーザーが記事を投稿すると、他のユーザーからのコメントがつく」アプリを想像してください。

あなたはコメント一覧画面を作っていましたが、新要件として「コメント一覧の取得が遅いので、画面表示前にプリフェッチしてほしい」という要望が来ました。

なんと嬉しいことに画面の処理とデータ取得処理が切り離されており、`ArticleUseCase`に`fetchComments()`があったので、それを画面表示より前のタイミングで呼ぶだけの簡単な修正でした。

しかしリリース後、バグ報告が上がりました。「まだ表示されていないコメントが既読になってしまう」というのです。

実はコメントには既読管理という概念がありました。`fetchComments()`の処理時に、サーバに既読フラグを送信していたのです。一覧画面表示時にこのメソッドを呼ぶ前提であればユースケースに沿った処理といえますが、その前提がメソッド名からは読み取れなかったために起こった悲劇でした。

「それはいまの文脈上どういう役割があるのか」「どう使われるのか」「使うと何が起こるのか」が正しく表現されていないメソッド名は、間違った使い方を誘発します。

Swiftコミュニティは公式に「API Design Guidelines[注8)]」を公開しています。

ガイドラインの冒頭には、API（Application Programming Interface：すなわちメソッド、型、変数など、プログラムで使われるすべてのインターフェイス）が守るべき3つの原則が明記されています。

- **使い道を明確に**
- **短い名前よりも明確であることが重要**
- **ドキュメントコメントを書く**

原則に従い、今回の処理を正確に表現するなら、こうなるでしょうか。

```
fetchCommentsAndMarkAllCommentsAsRead()
```

ひどく長い名前ですが、もう間違って使われることはないでしょう。原則「短い名前よりも明確であることが重要」が指摘するとおり、不正確な名前よりはマシです。

ところでガイドラインは、3原則の直後に注意事項として「シンプルに機能を言い表せない場合、設計を間違えているかもしれない」と述べています。名前が適切に表せないのは「メソッドの役割が大きすぎる」「適切な粒度での抽象化ができていない」ことのサインだと考えるのです。

今回の例であれば、メソッドを`fetchComments()`と`markAllCommentsAsRead()`に分けてみます。すると、「プリフェッチの場合は`fetchComments()`を呼ぶ」「画面表示されたタイミングで`markAllCommentsAsRead()`を呼ぶ」というように正しく処理できるようになりました。

正しい名前に向き合うことで、論理の破綻、隠れている役割、所属のおかしさを浮かび上がらせ、設計の改善に繋げられました。

注8）Apple Inc. (2018)『API Design Guidelines』https://swift.org/documentation/api-design-guidelines/

メソッドが所属する型は、無造作に突っ込まれた手続き処理の塊であってはいけません。明確な責務を表現する**データと振る舞いのセット**であるべきです。

名前から逃げないようにしてください。うわべだけきれいに整えても、問題が隠蔽されるだけです。現実を直視しなければ、よりよい設計には繋がりません。

なお、API Design Guidelinesが説いているのは、紹介した3原則だけではありません。丁寧に述べられた指針は、Swiftでコードを書くならば必ず守るべきものです。洗練・標準化されたネーミングのために必ずご参照ください。

2.2 アジャイル開発と設計の原則

名前に向き合う覚悟ができたら、そこがようやく設計の入口です。

責務を明確に示せてこそ、それぞれの型同士をどのように組み合わせるか——つまり、パターンをどうやって使うか、という話に入れます。ですが、パターンの話まではもう少しお待ちください。

2.2.1 アジャイル開発とは

2001年、名声のあるソフトウェア開発者17人が**アジャイル開発**という開発手法を提唱しました。第1章で整理したような近年のアプリ開発の状況によって、私たちの開発現場には「迅速に完成させ」「変化に対応できること」が求められています。アジャイルとは「俊敏」という意味です。アジャイル開発の目的はiOSアプリ開発の現場のニーズと合致しており、多くのiOSアプリ開発の現場でもこのプラクティスが採用されています。

アジャイル開発の提唱者のひとりであり、書籍「アジャイルソフトウェア開発の奥義」の著者でもあるRobert C. Martin氏（以降、彼の愛称からUncle Bob）は、アジャイル開発は次のようなサイクルであると述べています[注9)]。

1. アジャイルのプラクティスに従って問題を発見
2. 設計の原則を適用して問題を分析
3. 適切なパターンを適用して問題を解決

注9） Robert C. Martin (2008)『アジャイルソフトウェア開発の奥義 第2版』瀬谷啓介訳、p.118、SBクリエイティブ

第 3 章

Swiftらしく設計する

関 義隆／@takasek

第2章「設計にパターンを適用する前に」では次のことを説明しました。

- 名前によって責務の境界を明示する
- 不適切な責務（コードの臭い）は、設計の原則によって言語化できる
- 必要に応じて設計の原則、パターンを適用するが、やりすぎは禁物
- 設計は単純に始め、リファクタリングを進めながらイテレーティブに進化させる
- リファクタリングのためには、テストしやすい設計が前提

本章では具体的なコードを見ながら、設計の原則やSwiftの言語機能によってどのように設計を改善していけるかを説明します。

3.1 設計改善前のコード

あるアプリ内に「メッセージ」と呼ばれる概念が3種類あるとします。

- **TextMessage**：textを持つ
- **ImageMessage**：imageを持ち、オプショナルでtextを添えられる
- **OfficialMessage**：受信専用。ユーザーは送信不可能

そのようなメッセージを送信する際に使う型を考えます。責務は次のとおりです。

- メッセージ送信の元となる入力値を保持し、その値をサーバに送信する
- 通信結果を保持し、delegateに結果を伝える

これらの責務をもつ型`MessageSender`の、設計改善前のコードを示します。

● リスト 3.1　MessageSender.swift（改善前）

```
final class CommonMessageAPI {
    func fetchAll(ofUserId: Int,
                  completion: @escaping ([Message]?) -> Void) { ... }
    func fetch(id: Int,
               completion: @escaping (Message?) -> Void) { ... }
    func sendTextMessage(text: String,
                         completion: @escaping (TextMessage?) -> Void) { ... }
    func sendImageMessage(image: UIImage, text: String?,
                          completion: @escaping (ImageMessage?) -> Void) { ... }
}
final class MessageSender {
    private let api = CommonMessageAPI()
    let messageType: MessageType
    var delegate: MessageSenderDelegate?

    // MessageType.official をセットするのは禁止！！
    init(messageType: MessageType) {
        self.messageType = messageType
    }
    // 送信するメッセージの入力値
    var text: String? { // TextMessage,ImageMessage どちらの場合も使う
        didSet { if !isTextValid { delegate?.valid ではないことを伝える () } }
    }
    var image: UIImage? { // ImageMessage の場合に使う
        didSet { if !isImageValid { delegate?.valid ではないことを伝える () } }
    }
    // 通信結果
    private(set) var isLoading: Bool = false
    private(set) var result: Message? // 送信成功したら値が入る

    private var isTextValid: Bool {
        switch messageType {
        case .text: return text != nil && text!.count <= 300 // 300 字以内
        case .image: return text == nil || text!.count <= 80 // 80 字以内 or nil
        case .official: return false // OfficialMessage はありえない
        }
    }
    private var isImageValid: Bool {
        return image != nil // image の場合だけ考慮する
    }
    private var isValid: Bool {
        switch messageType {
        case .text: return isTextValid
        case .image: return isTextValid && isImageValid
```

```
        case .official: return false // OfficialMessage はありえない
        }
    }
    func send() {
        guard isValid else { delegate?.valid ではないことを伝える () }
        isLoading = true
        switch messageType {
        case .text:
            api.sendTextMessage(text: text!) { [weak self] in
                self?.isLoading = false
                self?.result = $0
                self?.delegate?. 通信完了を伝える ()
            }
        case .image:
            api.sendImageMessage(image: image!, text: text) { ... }
        case .official:
            fatalError()
        }
    }
}
```

外に公開されたインターフェイスは限られています。MessageSender はメッセージ送信に必要な機能をうまく隠蔽しているように見えます。

しかしこの型には多くの問題があります。

3.2 複数の責務を別の型に切り分ける

問題のひとつは、この型が複雑なバリデーションロジックを抱え込んでいる点です。

● リスト 3.2　MessageSender 内のバリデーションロジック

```
private var isTextValid: Bool {
    switch messageType {
    case .text: return text != nil && text!.count <= 300 // 300 字以内
    case .image: return text == nil || text!.count <= 80 // 80 字以内 or nil
    case .official: return false // OfficialMessage はありえない
    }
}
private var isImageValid: Bool {
    return image != nil // image の場合だけ考慮する
}
var isValid: Bool {
```

```
    switch messageType {
    case .text: return isTextValid
    case .image: return isTextValid && isImageValid
    case .official: return false // OfficialMessage はありえない
    }
}
```

本当に正しく動くのか、ひと目では判断できません。

isValidには引数がありません。どういう条件でisValidが変化するのでしょうか。依存しているプロパティは暗黙的で、コードを丹念に追わないと分かりません。しっかり読めば、依存しているのはmessageTypeと各種の入力値だと分かりますが、TextMessageの場合とImageMessageの場合で条件は変わります。新しいメッセージ種別や新しい条件が増えたら、ますますコードは複雑になることでしょう。

なぜこれほど複雑なことになってしまっているのでしょう。それほど込み入ったバリデーションではないはずですが…。コードから次のような臭いが漂ってきました。

- **硬さ**：変更しにくいシステム。1つの変更によってシステムの他の部分に影響が及び、多くの変更を余儀なくさせるようなソフトウェア
- **もろさ**：1つの変更によって、その変更とは概念的に関連のない箇所まで壊れてしまうようなソフトウェア
- **扱いにくさ**：正しいことをするよりも、誤ったことをするほうが容易なソフトウェア
- **不必要な繰り返し**：同じような構造を繰り返し含み、抽象化してまとめられる部分がまとまっていないソフトウェア

臭いを感じたら、なぜそう感じたのか考えてみます。おそらくなんらかの設計の原則に反しているのでしょう。

第2章で、コードの臭いを消臭するためには設計の原則が使えると述べました。有名な設計の原則には**SOLID原則**があります。S、O、L、I、Dそれぞれの頭文字を冠する5つの原則をまとめたものです。

ここではSOLID原則のS、**単一責任原則（Single Responsibility Principle）**がヒントになりそうです。

単一責任原則（Single Responsibility Principle）

- クラス（型）を変更する理由はふたつ以上存在してはならない

単一責任原則は**凝集**という概念を拡張したものだといわれています[注1)]。モジュール内での責務がまとまっている状態を、凝集度が高い状態と表現できます。

単一責任原則について勘違いしやすいのが、判断の基準になるのが**変更の理由**であるという点です。凝集という観点から言い換えると、手続きや利用シーンが近い機能がひとつのモジュール内に凝集

注1）Robert C. Martin（2008）『アジャイルソフトウェア開発の奥義 第2版』瀬谷啓介訳、p.121、SBクリエイティブ

していることは開発のしやすさにつながりますが、再利用性は落ちます。開発のしやすさと再利用性はトレードオフであり、そのバランスはプロジェクトの時期・要件によって流動するものです。変更可能性のない責務を無意味に切り離すことは、逆に「不必要な複雑さ」に結びつくおそれがあります。

変更の理由は、変更の必要性が生じたときにはじめて「理由」となりえます。単一責任原則を適用すべきか判断するには、その点の検討を忘れてはなりません。

3.2.1 単一責任原則を適用する

今回単一責任原則を適用すべきか、考えてみましょう。

バリデーションは十分に大きな関心事です。しかしそれはMessageSender本来の目的である「メッセージ送信」とは別の関心であり、別の型として扱うべきだという可能性を示唆しています。

凝集度について確認してみると、今回の例によく似た説明がありました[注2)]。

> **手続き的凝集 (Procedural Cohesion)**
>
> ■ **ある種の処理を行うときに動作する部分を集めたモジュール（たとえば、ファイルのパーミッションをチェックするルーチンとファイルをオープンするルーチンなど）**

ここからも、MessageSenderは典型的な**手続き的凝集**の罠を踏んでしまったのだと分かります。

「変更の理由」という観点はクリアしているでしょうか？

メッセージ種別が増えたりバリデーション条件が変わることは十分にありえます。そのようなときに困りそうなので、今は見通しが悪いコードをリファクタしたいのです。そう感じることも立派な変更の理由です。

やはりこの臭いの元は、単一責任原則に反していることのようです。

それではMessageInputValidatorという型を作って、そこにバリデーションのロジックを閉じ込めることにしましょうか。

● リスト 3.3　MessageInputValidator.swift

```
struct MessageInputValidator {
    let messageType: MessageType
    let image: UIImage?
    let text: String?

    var isValid: Bool { ... }
    private var isTextValid: Bool { ... }
    private var isImageValid: Bool { ... }
}
```

注 2)『凝集度 - Wikipedia』https://ja.wikipedia.org/wiki/%E5%87%9D%E9%9B%86%E5%BA%A6

第４章

アーキテクチャのパターンを鳥瞰する

関 義隆／@takasek

第２部ではいよいよ、MVC、MVP、MVVMなどといったアーキテクチャパターン[注1)]を解説します。

百花繚乱のアーキテクチャパターンには、それぞれが生まれた背景があります。各アーキテクチャの解決したい問題は異なります。文脈を辿ることで、各アーキテクチャの関係が見えてきます。

4.1 はじめに：２種類のアーキテクチャ

アーキテクチャを語るにあたり、まず整理しておくべきことがあります。一般にiOSアプリのアーキテクチャと呼ばれているものには、切り口の違う２種類のものが混在しています。本書ではそれを**GUIアーキテクチャ**と**システムアーキテクチャ**と呼びます。

MVC、MVP、MVVMといった耳馴染みのあるアーキテクチャには、MとVという文字が含まれます。これが**Model**と**View**を表すことはご存知の方も多いでしょう。しかし改めて考えると、Modelとは、Viewとは、一体何なのでしょうか。

これらのアーキテクチャの根底にあるのは**Presentation Domain Separation**（PDS）と呼ばれるアイデアです。「プレゼンテーション」「ドメイン」という単語は、それぞれ「View」「Model」とほぼ同じ意味だととらえて問題ありません。プレゼンテーションあるいはViewと呼ばれているのは**UIに関係するロジック**のことです。また、ドメインあるいはModelというのはアプリケーションのユーザーが思い描く、**システム本来の関心領域**です。

まとめると、「システム本来の関心領域（ドメイン）を、UI（プレゼンテーション）から引き離す」というのがPDSというアイデアです。そしてPDSを実践する際の具体的なレイヤー構造をパターンとして示すのが**GUIアーキテクチャ**と呼ばれるものなのです。

ただ、PDSによってきれいに関心を分離できたように見えるのは、あくまでViewからの視点でしかないことに注意してください。というのも、システム全体を俯瞰的に見ると「UIにもシステム本来の関心にも該当しない処理」が存在するからです。たとえば「サーバAPIからのデータ取得を試み、そこで発生したネットワークエラーをハンドリングする」「データをストレージに永続化する」といったデータ入出力にまつわる煩雑な処理はシステム本来の関心とは言いがたいでしょう。

GUIアーキテクチャの文脈でのModelの正体は、実際には「UIに関係しない処理すべて」と形容すべき、ざっくりした存在です。Modelをもっと具体的な何かだと思い込んでしまうのは典型的な

注1） 第１章で述べたとおり、本書においてはアーキテクチャとは「アプリの大まかなレイヤー分割のとらえ方」だと定義します。

過ちです。UIが関わる世界より先のことは、GUIアーキテクチャは何も示していないのです。

ではGUIアーキテクチャよりも広い――UIという単位にとらわれず、システム全体の構造を示すものは何かというと、それが**システムアーキテクチャ**と呼ばれるものです。

システムアーキテクチャについての詳細な説明は後に回し、まずGUIアーキテクチャについて見てみましょう。

4.2 GUIアーキテクチャ

iOSアプリに限らず、UIのあるシステムは基本的に次のような処理を行います。

1. UIが入力を受ける
2. 入力イベントをシステムが解釈し、処理
3. 処理の結果をUIに描画

UI（プレゼンテーション）とシステム（ドメイン）には明確な関心の違いがあります。プレゼンテーションは、どのようにコンポーネントをレイアウトし情報を装飾するかに興味があります。ドメインは、情報をどのようにモデリングし処理するかに興味があります。関心が違うならばレイヤーとして切り分けるべきだ、というのがPDSのコンセプトです。

Martin Fowler氏は、プレゼンテーションとドメインのレイヤーを分離するいくつかのメリットを挙げています[注2)]。

まず、単純に理解しやすいです。iOSアプリ開発者であれば誰しも、ドメインにあるべきロジックが紛れ込んだ、いわゆるFat View Controllerの洗礼を受けた覚えがあることでしょう。

次のメリットは、重複コードの排除です。複数のView Controllerに同じロジックに基づいた条件分岐を書いてしまったことはないでしょうか。逆に、同じようなレイアウトのViewを複製したことはないでしょうか。両者を分離し抽象的に扱えば、別々の画面でロジックを共通化したり、見た目が同じ画面に異なる情報を表示できます。

また、分業もしやすくなります。iOSでいえば、Auto Layoutや`UICollectionView`のFlow Layoutを適切に扱うにはそれ相応のスキルが必要です。一方、サーバから取得したデータの整形、イベントを受けて変化する状態の管理などは、まったく別のスキルが求められます。メンバーのスキルセットによって作業を分担するなら、手を入れるファイルやクラス自体が分かれていたほうがよいでしょう。

そして最後のメリットは、テスタビリティの向上です。プレゼンテーションのテストは困難です。`UILabel`の中身を覗いたり、描画されたViewの個数を数えるテストを書くのは苦行です。UIKitに依存しない静的なデータ構造がドメインとして形作られていれば、ユニットテストを書きやすくなります。

注2） Martin Fowler (2003)『PresentationDomainSeparation』http://martinfowler.com/bliki/PresentationDomainSeparation.html

このような明確なメリットがあるため、PDSはUIをもつあらゆるシステムを構築する際の基礎概念として浸透していきました。

そんなPDSの体現者であるGUIアーキテクチャの始祖が、**Model-View-Controller（MVC）**です。

4.2.1 Model-View-Controller（MVC）

原初MVCは、ノルウェーのオスロ大学の教授、Trygve Mikkjel Heyerdahl Reenskaug（以後Reenskaug）氏によって、Smalltalk開発環境のために考案されました。

当時の単純なGUIの開発ツールでは、**フォーム**に**ウィジェット**を貼り付けるのが普通でした。フォームというのは私たちでいうところのView Controllerのようなもので、レイアウトやUI全体のハンドリングを担当する概念です。ウィジェットというのはUIコンポーネント、つまりボタンやプログレスバー、テーブルなどを示します。

当時のコードはほとんどの場合、ウィジェットの中に直接書かれました。ウィジェットを利用する側から見ると、ウィジェット内部で何が行われているか気にせずにレイアウトできるため、それがよい分割だとされていたのです。

しかしReenskaug氏はこれを問題だと考えました。コードがウィジェットの表示上の都合に引きずられ、ユーザーのメンタルモデルと乖離してしまうためです。

1979年に彼はUIの表示を**Editor**、メンタルモデルを**Model**と名付け、その責務を分離しました[注3)]。

その後Editorの中でも入力に属するものと出力に属するものを分離するアイデアが生まれ、前者は**Controller**、後者は**View**と呼ばれるようになりました[注4)]。

Model、View、Controller。役者が揃いました。このアーキテクチャはSmalltalk-80の開発環境に組み込まれる形で完成しました。各レイヤーの具体的な役割分担は論文で詳細に解説されています[注5)]。

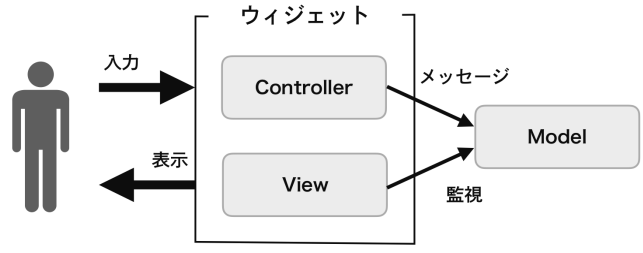
注3） Trygve Reenskaug（1979）『THING-MODEL-VIEW-EDITOR an Example from a planningsystem』http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf

注4） Trygve Reenskaug（1979）『MODELS - VIEWS - CONTROLLERS』http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf

注5） Glenn E. Krasner、Stephen T. Pope（1988）『A Cookbook for Using View-Controller User the ModelInterface Paradigm in Smalltalk-80』https://www.ics.uci.edu/~redmiles/ics227-SQ04/papers/KrasnerPope88.pdf

MVC のレイヤーの役割と構造

それぞれの役割と構造について解説します。



● 図 4.1　MVC の構造

● 表 4.1　MVC の各レイヤーの役割

レイヤー	役割
Controller	ユーザーの入力を受け、Model にコマンドを送る
Model	コマンドを受けて処理を行い、自身を更新
View	Model の変更を監視し、自身を更新

ControllerとViewによって構成されるウィジェットが、保持するModelにコマンドを送り、その変更を自身に反映するというのが基本の構造です。コマンドとは、単なるセッタ・ゲッタへの操作ではなく、ドメイン知識を元にした抽象度の高いものになります。

Modelはユーザーの**メンタルモデル**です。ユーザーは、アプリケーションが解決すべき問題に対して「このような構造である」という理解をもっています。Modelはそれに対応した構造になっていなくてはなりません。

Modelが「単純なオブジェクトひとつひとつのこと」であると考えるのはよくある誤解です。「オブジェクトをどのように使うか」というロジックや構造もまたModelの一部であり、そのようなロジックをModelの外に漏らしてしまう状態を、後にMartin Fowler氏は**ドメインモデル貧血症**[注6)]と呼びました。

ModelからViewへの変更通知はObserverパターンによって実現されます。なぜならModelは複数のViewやControllerに共有されうるものだからです。

まとめると、登場当初のMVCの特色は次の２点であるといえます。

- ウィジェット単位でプレゼンテーションとドメインを分離する
- Modelの変更に対し、オブザーバー同期が行われる

注 6)　Martin Fowler (2003)『AnemicDomainModel』https://martinfowler.com/bliki/AnemicDomainModel.html

ここで「登場当初」と但し書きしたのは、「ウィジェット単位」であることはMVCのその後の進化まで見据えると疑問符がつくためです。2002年のA. Knight氏とN. Dai氏の論文[注7]では、より高次のControllerが出現し市民権を得ている事実が確認できます。ウィジェット単位のControllerは**Input Controller**、高次のMediatorのようなControllerは**Application Controller**と呼ばれ、区別されます。

ただしアーキテクチャの歴史を辿る上ではMVCのControllerがウィジェットを単位としている点が重要になってくるため、今はその特徴を強調しておきます。

Column. MVCと他プラットフォーム

ここまで読んで「自分の知っているMVCと違う…」と意外に思う読者もいらっしゃるかと思います。

アーキテクチャの歴史を理解する上で重要なのは、アーキテクチャは少なからず生まれた開発環境に依存しているということです。

MVCはそもそもSmalltalk開発環境のGUIサポートを目的としていました[注8]。SmalltalkではGUIツール上で、スクリーンの各要素にViewとControllerのペアを設定できたのです。そのため、Smalltalkという環境を離れるとMVCのアイデアがうまく機能しません。さまざまな環境に移植される際、MVCは本来のコンセプトをそのまま使えず変形を余儀なくされました。

後の節で登場するMVPも環境の違いがきっかけで生まれました。また1999年にサーバサイドに適用された**MVC Model2（JavaServer Pages Model 2 architecture）**[注9]にはObserverパターンは登場しません。私たちになじみ深い**Cocoa MVC**（Appleが提唱しているMVC）も大きくコンセプトが異なります。「Cocoa MVCとは何なのか」については、後の節で詳しくご説明します。

MVCの課題

MVCにはいくつかの課題がありました。

- **プレゼンテーションロジックの表現ができない。**年齢を表すModelを考えます。あるViewでは「未成年なら青文字・成年なら赤文字」というロジックがあり、別のViewでは「還暦以上なら銀文字、それより若ければ黒文字」というロジックがあったとします。このロジックはどこで表現すべきでしょうか
- **プレゼンテーション状態を保持できない。**iOSでいうと「TableViewのセルの選択状態」や「すべてのフォームを埋めることでenabledになるボタン」などを、どのように管理するのでしょうか
- **テストが難しい。**当時のSmalltalkの実装では、Modelの変更を確認するためには具体的なViewインスタンスが必要でした。ウィジェットを前提にしていたからだと思われます

注7） A. Knight、N. Dai（2002）『Objects and the Web』http://translatedby.com/you/knight-and-dai-objects-and-the-web/original/
注8） 『Model View Controller History - C2 Wiki』http://wiki.c2.com/?ModelViewControllerHistory
注9） Govind Seshadri（1999）『Understanding JavaServer Pages Model 2 architecture』http://www.javaworld.com/article/2076557/java-web-development/understanding-javaserver-pages-model-2-architecture.html

第2部

iOSアプリのための設計パターン

第 5 章

MVC

史 翔新／@lovee

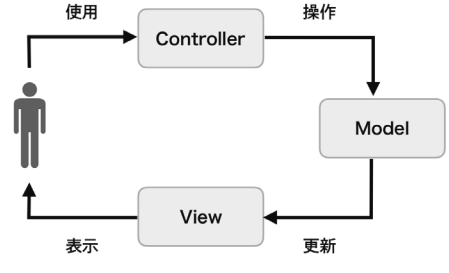
本章では、多くのGUIアーキテクチャ[注1)]の元祖ともいえる原初MVCパターンの概要と、Cocoa MVCについて解説します。

5.1 MVCとは

プログラムを「入力」「出力」「データの処理」の３つの要素に分けたとき、それぞれを**Controller**、**View**、**Model**と定義します。この３つがMVC（Model-View-Controller）を構成する要素となります。

MVCが登場した当時のインターフェイスでは、ユーザーは次のような操作をしていました。

1. キーボードやマウスといった入力デバイスで操作を行う。
2. 入力を検知したControllerがModelに何かしらの処理を依頼する。
3. 処理の結果がViewに伝わる。
4. モニターなどの出力デバイスに反映される。



● 図 5.1　MVC 登場当時の代表的なデータフロー

登場する要素とデータフローを整理したのが図5.1です。

ユーザーが入力デバイスを使用するとControllerが反応し、Modelを操作します。Modelが操作

注 1）　GUI アーキテクチャの多くで「Model と View と何か」という組み合わせが多いため、これらのアーキテクチャを「MVx」と表現することがあります。

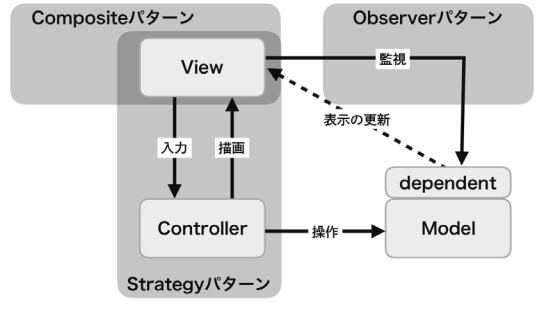
されることでModelの状態が更新され、同時にその内容をViewによってユーザーに示します。アプリケーションはこのサイクルを繰り返して動作します。

アプリケーションの処理から入力と出力とを分離・独立させたことで、プログラムの本質である「データ処理」そのものに専念しやすくなります。このような抽象化を施したアーキテクチャパターンが「MVC」です。

5.1.1 原初 MVC

原初MVCは、Composite、Strategy、Observerという3つのデザインパターンを中軸に構成されています。

ViewはCompositeパターンを用いて複数の子Viewを持つ集合体として振る舞い、さらにStrategyパターンを用いて適切なControllerを生成します。そして、ModelはObserverパターンを用いて自分自身の状態の変化を関連するViewに通知します。それぞれの関係は図5.2のとおりです。



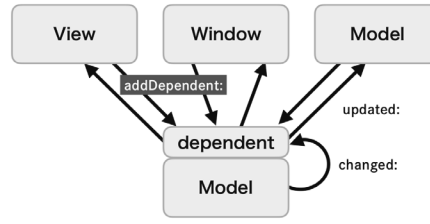
● 図 5.2 原初 MVC

ViewはModelの状態をObserverパターンで監視します。そしてViewはStrategyパターンによって、適切なControllerを必要に応じて生成します。Controllerは入力に応じてModelに処理を依頼します[注2)]。依頼を受けたModelは何かしらの処理を行い、自分の状態を更新します。ViewはModelの状態変化を監視しているため、更新を受け取ったViewはそれに応じて自身の描画を行います。

この通知のしくみとして、原初MVCを最初に実装したSmalltalk[注3)]では**Dependents**という技術を利用します。これはSmalltalkが提供するObserverパターンの技術のひとつです。

図5.3のように、ModelはDependentsという配列を持ち、状態更新を受け取りたいオブジェクトは自身をそのDependentsに登録します。

注 2） Controller はマウスイベントなどのインタラクションに際し、他の View と Controller のペアとのやりとりをすることもあります。
注 3） 1980 年に公開された純粋オブジェクト指向プログラミング言語、およびそれによって構築された統合開発環境です。オブジェクト指向のパイオニア的な存在として、Objective-C をはじめ後世に大きな影響を与えました。



● 図 5.3　Smalltalk の Dependents のしくみ

そして、更新されたタイミングでModelにchanged:メッセージを送ると、Observerパターンによって Dependents 内のすべての部品に対してupdate:メッセージが送られます。この Dependents には、ViewやWindowもしくは別のModelを登録できるため、Modelは登録されたオブジェクトの具体的な実装を知ることなく更新されたことを通知できます。

Column.　Composite/Strategy/Observer デザインパターン

本文中で登場したComposite/Strategy/Observerの3つのデザインパターンは、それぞれ次のようなしくみを提供します。

Compositeパターン

再帰的・階層的なデータ構造を実現するデザインパターンです。Viewは複数のViewを自身の子供として持て、それらの子供もまた同じように子Viewを持てます。そして、これらすべてのViewは「Viewとして」振る舞えます。

Strategyパターン

インターフェイスのみ定義しておき、具体的な実装は実行時に選べるデザインパターンです。たとえば、ControllerはModelやViewに必要なメソッド名だけ定義し、実行時にModelやViewをさまざまな物に置き換えても同じメソッドで動かせます。

Observerパターン

イベントを監視し、発火したイベントに応じた処理を実現するデザインパターンです。たとえばViewはModelが持つ特定プロパティの変更を監視し、最新の値を自分の表示処理に流し込めます。

5.1.2 Cocoa MVC

原初MVCの登場から時代も進み、アプリケーションの様相も大きく変わりました。より複雑なGUIやタッチスクリーンの登場により、当時明確に区別されていた「入力」と「出力」の境界線が、実はあいまいであることが顕在化しました[注4)]。また、ますます複雑になってきたさまざまな機能をうまく使えるように画面を組むため、一貫性のあるUIデザインが求められるようになりました。当然、Viewの再利用性を高めたり、ViewとModelを完全分離したいといった要望も出てきました。そのような背景の中、誕生したのがAppleの提唱したCocoa MVCです[注5)]。

Cocoa MVCの大きな特徴は、ViewとModelが完全に独立しているところです。原初MVCではViewがModelを監視するしくみでしたが、このしくみではViewは特定のModelに依存する作りになりがちです。見た目が同じようなViewであっても、それぞれ参照するModelが違ってくれば、何かしらの仕様変更が必要となり、View自体の再利用性が下がってしまいます。

そこでAppleは、ViewもModelのように再利用性の高いコンポーネントにするために、View-Model間の関係を切り離し、両方をControllerが参照するようにしました。これによってViewはModelを監視する必要がなくなり、見た目が同じようなものなら、どんなにModelが変わろうともViewを変更しなくてすみます。もちろん従来のMVCと同じように、Viewがどれだけ変わろうとModelは何も修正する必要はありません。

もちろんCocoa MVCにもデメリットはあります。Controllerが他のすべてを参照するので、ModelとViewに比べてControllerの再利用性が下がってしまうことです。しかしAppleはそれを踏まえた上で、ModelとViewを完全に分離することが重要だと謳っています[注6)]。したがってCocoa MVCにおいては、Controllerの再利用性をあきらめた設計になることはやむをえません。

5.2 コードから見るMVC

ここからは、Viewをタップした際に、数字をカウントアップするような処理をそれぞれのMVCではどうやって実現するか、Swiftのコードで見てみましょう。

5.2.1 原初MVC

原初MVCはもともとSmalltalk上で考えられたアーキテクチャのため、Smalltalkの実装に大きく依存する部分があります。それでもあえてSwiftで実装することを考えます。

注4) たとえばボタンオブジェクトを考えてみるとわかりやすいでしょう。ボタンのタップ時の動作が入力でありながら、同時にボタンの見た目が出力になります。

注5) 正確にはAppleはこの設計に特定の名前を付けていませんが、公式ドキュメントでは「Cocoa MVC」だったり、「MVC (Cocoa)」だったりと複数の名前で登場します (https://developer.apple.com/jp/documentation/CocoaEncyclopedia.pdf)。

注6) アップルの公式開発ドキュメント「Objective-C プログラミングの概念」(p50) によると、「ビューやモデルは、アプリケーションでも特に再利用性の高いオブジェクトでなければなりません」と書かれています (https://developer.apple.com/jp/documentation/CocoaEncyclopedia.pdf)。

Model

Modelは複数のViewとControllerに自分自身の状態更新を通知する必要があります。Swiftの標準ライブラリにはSmalltalkのDependentsと同じしくみはありませんが、ここで実現したいのは「更新を複数のオブジェクトに通知する」ことです。そこで通知には`NotificationCenter`を使います。

● リスト 5.1　Model クラス

```
final class Model {
    let notificationCenter = NotificationCenter()
    private(set) var count = 0 {
        didSet {
            notificationCenter.post(name: .init(rawValue: "count"),
                                    object: nil,
                                    userInfo: ["count": count])
        }
    }
    func countDown() { count -= 1 }
    func countUp() { count += 1 }
}
```

`count`が更新されたときに`NotificationCenter`に通知を送ることで、監視できる状態を作り出しています。Controllerが`count`を更新するために、`countDown`と`countUp`のメソッドも必要です。

Controller

原初MVCのControllerは`UIViewController`ではないため、独自のクラスを定義します。

Controllerには Modelの`countUp()`と`countDown()`を実行する処理を実装します。入力に対してModelへ処理を依頼する必要があるため、Modelをプロパティとして保持します。しかしModelの更新通知は受け取りません。

● リスト 5.2　Controller クラス

```
class Controller {
    weak var myModel: Model?
    required init() {}
    @objc func onMinusTapped() { myModel?.countDown() }
    @objc func onPlusTapped() { myModel?.countUp() }
}
```

`onMinusTapped()`／`onPlusTapped()`メソッドは、Viewで定義する`UIButton`のタップイベントを`addTarget(_:action:for:)`によって直接受け取るため、`@objc`修飾子を付けています。イベントを受け取ったらModelに処理を依頼します。

第 6 章

MVP

田中 賢治／@ktanaka117, 加藤 寛人／@hkato193

6.1 MVP アーキテクチャ

MVPは画面の描画処理とプレゼンテーションロジックとを分離するGUIアーキテクチャです。本書ではMVPの歴史的な背景に触れつつ、Martin Fowler氏が自身のサイトにまとめたPassive ViewとSupervising Controllerという２つのパターンについて解説します。

MVPの目的はテスト容易性と作業分担のしやすさを手に入れることです。時代を追うごとに、アプリケーションの複雑化によって保守のしやすさがいっそう求められるようになりました。その目的を叶えるためにMVPがとったアプローチは、MVCの責務の再分割でした。

MVPにおけるPassive ViewとSupervising Controllerの２つのパターンの違いは、責務の分け方にあります。Passive Viewではプレゼンテーションロジックを完全に**Presenter**に担当させます。一方でSupervising Controllerでは複雑なプレゼンテーションロジックをPresenterに担当させつつ、簡単なものはViewに残し、Modelの状態変更通知を検知したらView自身でも、プレゼンテーションロジックを処理するという違いがあります。これらの具体的な違いについては、このあと詳しく紹介します。

さらに本章では、Passive Viewの実装イメージをもちやすくなるようサンプルコードとともに解説します。

6.1.1 MVP の歴史

まず第４章でも触れてきたMVPの歴史を軽くおさらいしましょう。

Potel氏の論文のMVPはSmalltalkで使われる古典的なMVCを一般化し、さまざまなアプリケーションやクライアント／サーバ形式のアーキテクチャにも対応できるようにした意欲的なものでした。しかしこのときはまだModel ／ View ／ Presenterの他にInteractorやCommandなどのコンポーネントがあり、今とは異なる形のアーキテクチャでした。

次に変化があったのはBower氏とMcGlashan氏のMVP[注1)]でした。Taligent社のMVPをシンプルにして、Model ／ View ／ Presenterの３つでアーキテクチャを表したものです。この時点で今のMVPに近い形となりました。

注 1)　http://esug.org/wiki/pier/Conferences/2000/Model-View-Presenter_-Twisting-the-triad

そしてFowler氏が自身のサイト[注2)]でMVPを整理し、「**Passive View**」と「**Supervising Controller**」の2つの派生をもつアーキテクチャとしてまとめたものが、現在よく知られているMVPです。

6.1.2 MVPの目的

MVPの目的はテスト容易性と作業分担のしやすさを手に入れることです。これは言い換えるならば、保守のしやすさとも言えます。より複雑なアプリケーションになるほど、保守のしやすさが重要になっていきます。

この目的に対して、MVPではPresenterというプレゼンテーションロジックを担うコンポーネントと、PresenterがViewに対して手続き的に描画指示を出すフロー同期[注3)]を導入して解決を図りました。

iOSにおけるMVPでは、必要な箇所をprotocolとして宣言してコンポーネント間を疎結合にすることで、テスト容易性と作業分担のしやすさを実現しています。

Column. MVCを細分化したMVP

Taligent社のPotel氏の論文では、MVPはMVCの責務を細分化したパターンとして紹介されました。具体的には、ModelとPresenterの間にはCommandsとSelectionsというコンポーネントが、ViewとPresenterの間にはInteractorというコンポーネントが存在していました。これらの細分化はテスト容易性や役割分担のしやすさを高めてくれる効果があり、大規模開発においては有効な手段のように思えます。

しかしその後、（少なくともiOSプラットフォームでは）名前を聞かなくなっている理由として、細分化のしすぎによる弊害があったからだと予想されます。多くの場合、開発対象に合わない責務の細分化をしてしまうと、開発者にとって負担になります。GUIにおいて、ここまでの責務の細分化が必要になる開発対象が多くはないということが、この源流がメジャーになっていない原因だと筆者は考えます。

注2） https://www.martinfowler.com/eaaDev/ModelViewPresenter.html
注3） フロー同期については「6.2 データの2つの同期方法」で解説します。

6.2 データの2つの同期方法

本章で紹介するMVPの２つのパターンを理解する上で重要になるのは、コンポーネント間のデータを同期する２つの方法です。それぞれ**フロー同期**と**オブザーバー同期**といいます。この２つについて、MVPの２つのパターン自体を解説する前に説明します。

6.2.1 フロー同期とオブザーバー同期

フロー同期（Flow Synchronization）とは、上位レイヤーのデータを下位レイヤーに[注4)]都度セットしてデータを同期する、手続き的な同期方法です。

オブザーバー同期（Observer Synchronization）は、監視元である下位レイヤーが監視先である上位レイヤーからObserverパターンを使って送られるイベント通知を受け取ってデータを同期させる、宣言的な同期方法です。

フロー同期がオブザーバー同期と比較して有利なのは、データフローを追いやすいことです。たとえばフロー同期が適する具体的な場面として、push遷移でドリルダウンしていくような、隣り合った画面間でデータを共有する例が挙げられます。画面がpush／popするタイミングで遷移先の画面にデータを同期すればよいため、データフローが追いやすくなります。隣り合った画面は、遷移先の画面（上位レイヤー）が遷移元の画面（下位レイヤー）の参照を持ちやすいため、フロー同期が有効です。

オブザーバー同期はデータが変更されるたびに同期処理が実行されるため、いつデータが同期されるかが追いづらくなるというデメリットを持っています。

オブザーバー同期がフロー同期と比較して有利なのは、共通した監視先を持つ複数の箇所で、データを同期しやすいことです。たとえばオブザーバー同期が適する具体的な場面として、よくハートマークや星マークで表現されるUIをもつ、お気に入りのデータを管理する画面が挙げられます。複数のタブや階層が離れた画面においても、それぞれの画面が共通のデータ領域の変更を監視しているため、同期箇所で他の画面の参照を持つ必要がありません。いずれかの画面でお気に入りのデータを書き換える操作をすれば、その通知を宣言的に監視するすべての画面でデータの同期が行われ、描画処理が発動します。

フロー同期は手続き的に同期を指示するため、共通したデータを参照しているすべての箇所の参照を持っておかなければいけなくなります。これは参照の管理が煩雑になりやすいというデメリットを持っています。

隣り合った画面同士はフロー同期が有効だと説明しましたが、これは画面に限らず、上位レイヤーが下位レイヤーの参照を得られる場合にフロー同期が有効ということです。データを同期するコンポーネント同士の間に、他のコンポーネントが挟まるほど、コンポーネント間の距離は遠くなりま

注 4） 上位レイヤーと下位レイヤーの概念は「3.3.2 依存関係を「逆転」するという意味」をご参照ください。

す。コンポーネント間の距離が遠い関係同士で無理なフロー同期をしようとすれば、上位レイヤーが知らなくてもよい下位レイヤーを知る（依存する）ことになり、データフローを追いやすいというメリットを殺してしまいます。

データの同期が画面同士でのみ適用されるわけではないというのは、オブザーバー同期にも当てはまります。オブザーバー同期は上位レイヤーが下位レイヤーの参照を持つ必要なく、かつコンポーネント間の距離に関係なくデータの同期ができるという点が、フロー同期よりも優れている点です。上位レイヤーが下位レイヤーの参照を持たなくてよいということは、互いに疎結合になるとも説明できます。

6.2.2 MVPにおけるデータの同期方法

MVPには、本章で説明するように**Passive View**と**Supervising Controller**の２つの方式があります。

Passive ViewはPresenter→View間にフロー同期を使います。iOSアプリ開発におけるMVPのフロー同期とは、`label.text = newText`による描画処理や、データソースを更新してから`tableView.reloadData()`を呼び出す描画処理などが例として挙げられます。

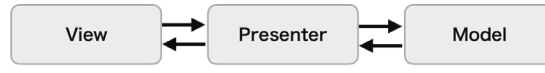
Supervising Controllerは両方の同期方法を使い、Presenter→View間をフロー同期し、Model→View間をオブザーバー同期します。フロー同期はこの直前で説明したとおりです。iOSアプリ開発におけるSupervising Controllerのオブザーバー同期とは、監視元であるViewが、監視先であるModelから`NotificationCenter`などで送られるイベントを受け取って、描画処理を行うことが例として挙げられます。

２つの同期方法はどちらが優れているというものではなく、それぞれにメリット・デメリットを持っているため、設計段階でどちらを採用するか考える必要があります。この２つの同期方法はMVPのみならず、多くのアーキテクチャパターンで共通して利用される方法です。場面ごとの有利不利をよく理解しておきましょう。

6.3 MVPの構造

Passive ViewとSupervising Controllerの構造について説明します。この2つは共通する事項も多く存在するため、本節では共通した構造に触れてからそれぞれについて説明します。

図6.1に示すとおり、MVPにはModel／View／Presenterという3つのコンポーネントが登場します。それぞれについて見ていきましょう。



● 図 6.1　MVP の基本的な構成 (Passive View)

6.3.1 共通事項

Model

ModelはUIに関係しない純粋なドメインロジックやそのデータを持ちます。画面表示がどのようなものでも共通な、アプリの機能実現のための処理が置かれます。MVPにおけるModelはMVC、MVVMにおけるModelと同じ立ち位置です。

第4章でも触れたとおり、MVPはGUIアーキテクチャに属するため、Modelの詳細な責務分けについては関知しません。Modelが扱う領域の具体例はWebAPIやデータベースへのアクセス、BLEデバイス制御や、会員ステータスごとの商品の割引率の計算など多岐に渡ります。

またModel自身は他のコンポーネントには依存していません。ViewやPresenterがなくてもビルド可能であることを表します。

View

Viewはユーザー操作の受け付けと、画面表示を担当するコンポーネントです。iOSのMVPにおいてはView ControllerもViewに含む解釈をします。

Viewはタップやスワイプなどによる UIイベントを受け付け、Presenterに処理を委譲したり、Modelの処理を呼び出したりします。

Modelに変更が発生したら、なんらかの方法でViewにそれが伝達され、表示内容が更新されます。更新方法の違いがPassive ViewとSupervising Controllerの違いになります。この違いについては、このあとで詳しく説明します。

第7章
MVVM

田中 賢治／@ktanaka117

7.1 MVVMアーキテクチャ

MVVMアーキテクチャは画面の描画処理とプレゼンテーションロジックとを分離するGUIアーキテクチャです。GUI構造をModel／View／ViewModelの3つに分け、画面の描画処理をViewに、画面描画のロジックをViewModelというコンポーネントに閉じ込めます。

そしてViewとViewModelを**データバインディング**と呼ばれるしくみで関連付けることで、ViewModelの状態変更に同期してViewの状態も更新され、画面に反映されることが特徴です。

このデータバインディングによって、プレゼンテーションロジックを担うViewModel内にViewに対する手続き的な描画指示を書く必要がなくなります。宣言的なバインディングにより、ViewModel自身の状態を更新するだけで、Viewの描画処理が発火するためです。

ViewとViewModelとはデータバインディングによって完全に疎結合となるため、具体的なViewが存在しなくてもプレゼンテーションロジックをテストしやすいメリットもあります。

Column.　データバインディング

データバインディングとは、2つのデータの状態を監視し同期するしくみのことです。片方のデータ変更をもう一方が検知して、データを自動的に更新します。バインディングの関係は1方向だけではなく、双方の変化を互いに監視することもあります。

本書ではViewがViewModelを監視する、単方向バインディングで解説します。

7.2 MVVMの歴史

MVVMは、当時MicrosoftのWPFおよびSilverlightアーキテクトだったJohn Gossman氏のブログ**注1)**で、2005年に発表されました。

Martin Fowler氏がプラットフォームに依存しないViewの抽象化を作成する手段としてPresentation Model[注2)]を導入し、Gossman氏が汎用的なPresentation ModelパターンをWPFおよびSilverlightプラットフォーム向けに特殊化したものがMVVMです。[注3)]

7.3 MVVMの目的

もともとのWPFにおけるMVVMのねらいと、iOSにおけるMVVM実装のねらいは異なります。双方を比較しながら紹介します。

7.3.1 WPFにおけるMVVM

WPFにおけるMVVMのねらいは2つあります。

ひとつは、UIの実装とロジックの実装を別々の開発者で作業分担できるようにすることです。

UIの実装者と、アプリケーションロジックの実装者とでは求められる専門性が違います。またそれぞれはXAML／C#という別々の言語を使います。データバインディングを使ってUIとロジックを疎結合にすることで、作業を分担しつつそれぞれで生産性を高められるようになります。

もうひとつは、責務の分離によって変更箇所を小さくすることです。

UIとアプリケーションロジックの開発サイクルは異なるため、責務の分離による変更箇所を小さくすることはコンフリクトを防ぎ、生産性の向上につながります。両方を同じ開発者が担当していても同様です。

7.3.2 iOSにおけるMVVM

iOSでMVVMを採用する多くの理由は、関数型リアクティブプログラミング（Functional Reactive Programming。以降FRPと記述）と相性が良いためです。

Swiftでは最近FRPに関する議論が活発です。FRPはModelの変更やViewの変更が相互接続されてリアクティブに反応しあえる点と、手続き的ではなく宣言的にロジックを表現できる点が、

注1） https://bit.ly/2OGU6kK
注2） https://www.martinfowler.com/eaaDev/PresentationModel.html
注3） https://msdn.microsoft.com/ja-jp/magazine/dd419663.aspx

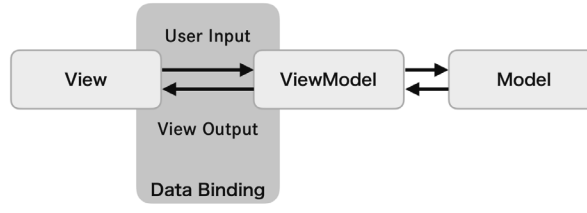
MVVMと相性が良いとされています。

Swiftでこうしたリアクティブな機能を実現するにはKVOやNotification Centerが使えますが、少し手続き的な記述になるのと、手間がかかるという難点があります。しかしこれらはRxSwift[注4]やReactiveSwifta[注5]などのFRPに特化したライブラリによって解決できます。ライブラリを使うことで、より宣言的な記述をしつつ複雑なバインディングを表現できます。

FRPに便利なライブラリの多くは、PromiseやResult型のようなロジックを書くのに役立つ機能を提供しています。もともとこうした表現を利用するためにライブラリを採用していて、さらにデータバインディングとも相性が良いことからMVVMを採用するという流れも多いのではないでしょうか。

7.4 MVVMの構造

MVVMは図7.1に示す3つのコンポーネントによって構成されています。



● 図 7.1　MVVM の基本的な構造

図中のModel、View、ViewModelそれぞれの役割を説明します。

7.4.1 Model

ModelはUIに関係しない純粋なドメインロジックやそのデータを持ちます。MVVMにおけるModelはMVC、MVPにおけるModelと同じ立ち位置です。

第4章でも触れたとおりMVVMはGUIアーキテクチャに属するため、Modelの詳細な責務分けについては関知しません。Modelが扱う領域の具体例はWebAPIやデータベースへのアクセス、BLEデバイス制御や、会員ステータスごとの商品の割引率の計算など多岐に渡ります。

またModel自身は他のコンポーネントには依存していません。ViewやViewModelがなくてもビルド可能であることを表します。

注 4）https://github.com/ReactiveX/RxSwift
注 5）https://github.com/ReactiveSwift/ReactiveCocoa

7.4.2 View

Viewはユーザー操作の受け付けと、画面表示を担当するコンポーネントです。

ViewはViewModelが保持する状態とデータバインディングし、ユーザー入力に応じてViewModelが自身が保持するデータを加工・更新することで、バインディングした画面表示を更新します。

7.4.3 ViewModel

ViewModelはView-Model間の画面表示のための仲介役であり、次の責務を担います。

- Viewに表示するためのデータを保持する
- Viewからイベントを受け取り、Modelの処理を呼び出す
- Viewからイベントを受け取り、加工して値を更新する

Modelはアプリケーションのビジネスロジックを知っていますが、それが画面上でどのように表示されるかを知っているべきではありません。ビジネスロジックと独立した、画面表示のために必要な状態とロジックを担うのが、ViewModelです。

Viewの抽象化を行なっている点でMVPのPresenterと役割のかぶる部分は多いものの、いくつか違いがあります。PresenterはViewに対して手続き的な更新処理を書かなければいけないため、Viewの参照を保持します。一方でViewModelはViewの状態と自身が持つ状態を関連づけること（＝データバインディング）によって状態を更新し、手続き的な更新処理を必要としません。そのためViewModelはViewの参照を保持しません。

7.5 実装から見るMVVM

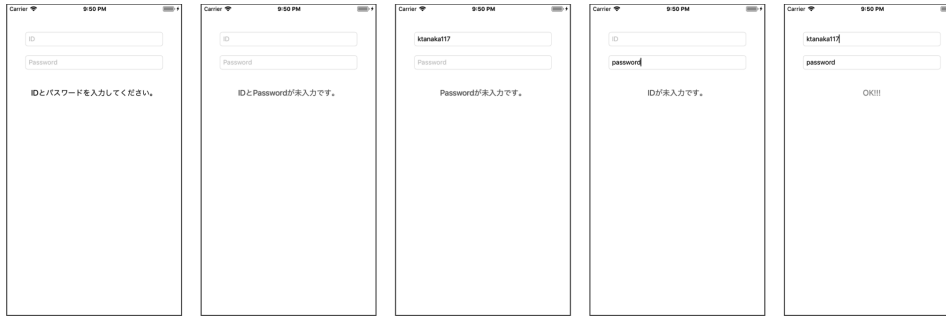
iOS上でMVVMを実装する方法はさまざまですが、本書では外部ライブラリを使わない実装として「Notification Centerによる実装」と、外部ライブラリを使った実装として「RxSwiftによる実装」の２つのパターンを、サンプルコードと共に解説します。

実現するのは、簡単なキー入力のバリデーション機能です。

- 画面上にはIDとパスワードの２つを入力するテキストフィールドが存在する
- IDとパスワードの両方に少なくとも１文字以上入っていなければならない
- どちらか一方でも文字入力がない場合、画面上にエラー文字列を表示する

IDとパスワードフィールドへの入力に応じてバリデーションを行い、満たされていないときはエラーを返却します。エラーの有無に応じてラベルの文字列と色が変化するようにします。

図7.2がアプリのスクリーンショットです。



● 図 7.2　バリデーションアプリのスクリーンショット

この機能をNotification CenterまたはRxSwiftを使って実装します。

ここで紹介するコードは本書のサンプルコード集に記載されています。ぜひ本書で解説している内容と比べながらお読みください。

7.5.1　Notification Center による実装

iOSでMVVMを実装する際は、一般的にFRPを実現するライブラリを使います。しかしここではMVVMのしくみを理解するためにもライブラリに頼らず、Notification Centerを使った実装方法を紹介します。

なおここで紹介する方法は、あえて馴染みのあるクラスを使って最低限のView-ViewModel間のデータバインディングを実現する方法です。手間も多く、あまり現実的な方法ではありません。実際には外部ライブラリを利用することがほとんどですので、データバインディングのしくみを知るための実装例としてとらえてください。

View

Viewの責務は次の3つです。

- ユーザー入力をViewModelに伝搬する
- 自身の状態とViewModelの状態をデータバインディングする
- ViewModelから返されるイベントを元に描画処理を実行する

それぞれ次のコードを元に説明します。

第 8 章
Flux

鈴木 大貴／@marty_suzuki

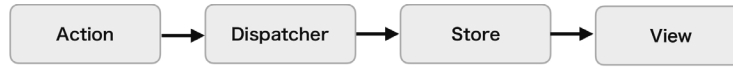
Fluxアーキテクチャは2014年のF8[注1)]で発表された、FacebookのWebアプリで利用されているGUIアーキテクチャです。Fluxアーキテクチャはアプリケーション開発においてスケールしやすいアーキテクチャを目指す形で誕生しました。Fluxアーキテクチャのコンセプトは、flux-concepts[注2)]にて公開されています。本章では、Webアプリで利用されているFluxアーキテクチャが、iOSアプリにどう適用できるか解説します。

8.1 Fluxアーキテクチャの概要

Fluxアーキテクチャは、ラテン語のFluxusという単語から名付けられています。Fluxusは**Flow（流れ）**を意味しており、その名のとおりFluxアーキテクチャでは**データフローが単一方向**であることが中心の考えになっています。

8.1.1 単一方向のデータフロー

Fluxアーキテクチャの中心的な考えである、単一方向のデータフローを表したのが次の図8.1です。



● 図 8.1 単一方向のデータフロー

ActionからViewにかけて、データが単一方向に流れていることがわかります。Fluxアーキテクチャの中心的なコンポーネントは、図8.1に示すAction・Dispatcher・Store・Viewの４つから構成されます。

注 1） Facebook Developer Conference の名称 : https://www.f8.com/
注 2） https://github.com/facebook/flux/tree/3.1.1/examples/flux-concepts

Action

実行する処理を特定するための`type`と、実行する処理に紐づく`data`を保持したオブジェクトです。

Dispatcher

Actionを受け取り、自身に登録されているStoreに伝えます。

Store

状態を保持し、Dispatcherから伝わったActionの`type`と`data`に応じて、状態を変更します。

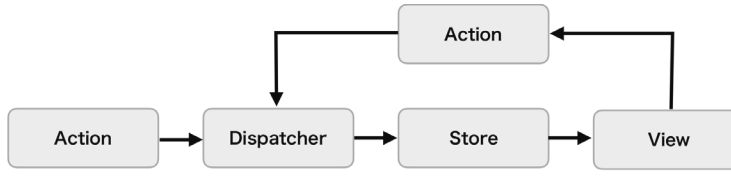
View

Storeの状態を購読し、その変更に応じて画面を更新します。

再度データフローを追います。DispatcherにActionを渡すことから始まり、DispatcherはActionをStoreに伝えます。Storeは受け取ったActionをもとに自身の状態を更新し、Storeの状態を購読していたViewが変更に応じて画面を更新します。繰り返しとなりますが、Fluxアーキテクチャでのデータフローは単一方向です。ViewからStoreへ直接変更を加えることはできません。

8.1.2 Viewから始まる単一方向の流れ

ViewからStoreへ直接変更を加えることはできないと説明しましたが、ViewからDispatcherに対してActionを渡すことでStoreに変更を加えることはできます。Viewから始まるデータフローを表したのが、次の図8.2です。



● 図 8.2　View から始まるデータフロー

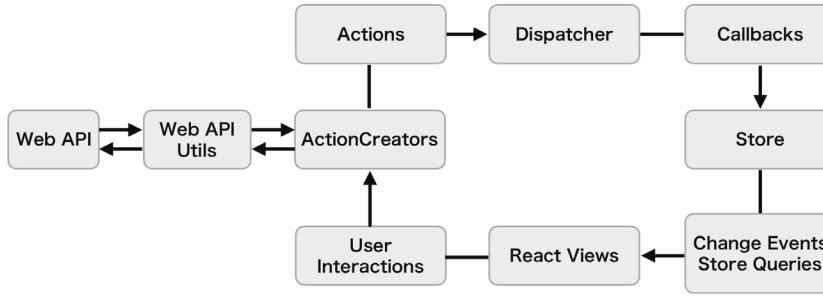
ユーザーの入力を受けたViewは、その入力をもとにActionを生成しDispatcherに渡します。Dispatcherは受け取ったActionをStoreに伝えます。StoreではActionをもとに状態が変更され、そのStoreの状態を購読しているViewが画面を更新します。このようにViewからデータフローが始まったとしても、Dispatcherがハブとなることでデータフローを単一の方向で実現できることがわかります。

Fluxアーキテクチャは Action・Dispatcher・Store・View の４つのコンポーネントと、データフローが単一方向という制約とで構成されます。この構成により、どこで状態の変更が起きているの

かの見通しが立てやすくなります。またそれぞれのコンポーネントが独立しているため、スケールしやすい構成になっています。

8.2 iOSアプリでの構成とデータフロー

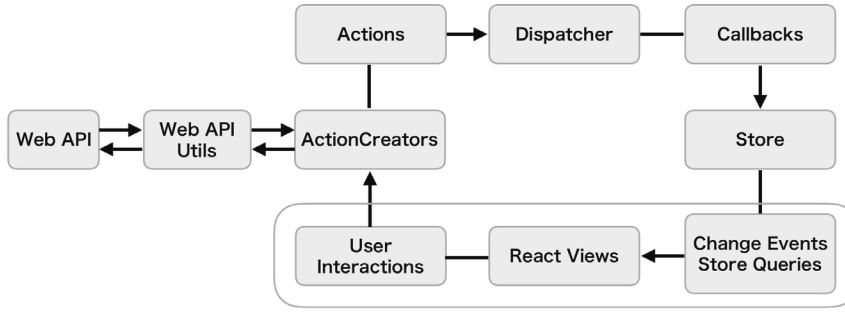
iOSアプリで実際にFluxアーキテクチャを採用する場合、どのような構成・データフローになるのでしょうか。まずはflux-todomvc[注3)]で公開されているWebアプリのサンプルを例に説明します。図8.3はWebアプリでFluxアーキテクチャを用いた場合の構成とデータフローです。



● 図 8.3　Webアプリで用いられる構成

8.2.1 Viewの構成とデータフロー

まずはViewコンポーネントについて説明します。図8.4で囲われた部分がViewコンポーネントの構成と前後のデータフローです。

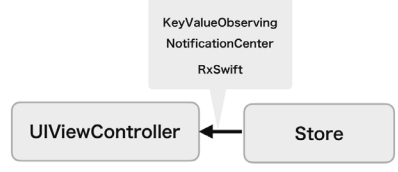


● 図 8.4　Webアプリで用いられる構成

注3)　https://github.com/facebook/flux/tree/3.1.0/examples/flux-todomvc

iOSの場合、UIViewControllerやUIViewがViewの役割を果たします。Viewから始まるデータフローは、ユーザーの入力によるユーザーインタラクションとして扱われます。たとえば、UIControlの`addTarget(_:action:for:)`で登録したメソッドなどからユーザーの入力を受け取ることで、Viewのデータフローが発生します。Fluxアーキテクチャでは状態を管理するのはStoreの役割であるため、Viewが状態を持つことはありません。Viewへ向かうデータフローは、Storeの状態をViewに反映するデータフローとなります。iOSアプリでは`NotificationCenter`**の通知機能**や**Observerパターンを担うライブラリ（RxSwift、ReactiveSwiftなど）**などでデータフローを実現できます。

iOSアプリにおけるViewコンポーネントの構成と前後のデータフローは、図8.5のように表せます。

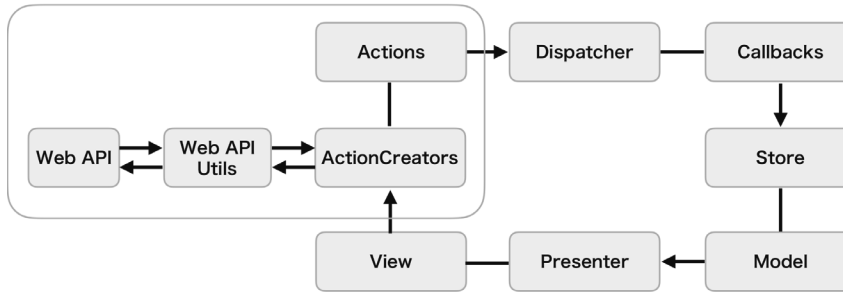


● **図 8.5　iOS アプリの View への反映**

iOSアプリの開発において、View Controllerが肥大化してしまう問題はよく起きます。Fluxアーキテクチャを用いるとView Controllerが状態を持つことはなくなり、状態に関するコードをView Controllerに記述する必要がなくなります。そして肥大化を避けられます。

8.2.2　Action（ActionCreator）の構成とデータフロー

Actionコンポーネントについて説明します。図8.6で囲われた部分がActionコンポーネントの構成と前後のデータフローです。



● **図 8.6　Web アプリで用いられる構成**

枠内のActionCreatorとは、次に示す役割をもったコンポーネントです。

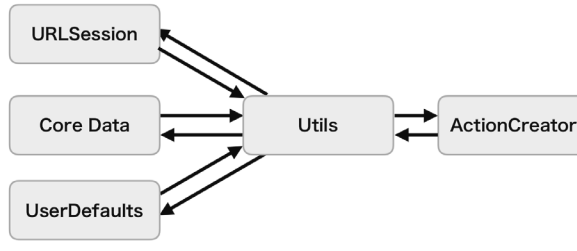
- 何らかの処理を行いその結果からActionの生成
- 生成したActionをDispatcherへ送信

データフローは、ユーザーの入力をもとにActionCreatorの処理を実行します。ActionCreatorはWeb API Utilを通してWeb APIから外部のデータを取得します。取得したデータをもとにActionを生成し、Dispatcherに送信します。

iOSアプリにおける、標準のフレームワークを利用してデータ取得する場合の主な候補は次の２つです。

- URLSessionでAPIサーバからデータを取得
- CoreDataやUserDefaultsからローカルデータを取得

ActionCreatorは、図8.7のように間にUtilを挟んだり、あるいは直接呼び出したりして外部のデータを取得します。



● 図 8.7　iOS アプリの Action コンポーネントの構成

もし「8.1.2　Viewから始まる単一方向の流れ」図8.2のように、ActionCreatorを使わずViewからActionを直接Dispatcherに渡した場合、「外部からのデータの取得」と「Actionを生成する」という処理がViewに記述されることになります。その場合View Controllerが肥大化する問題に直面してしまうので、注意しましょう。

第 9 章
Redux

杉上 洋平／@susieyy

本章ではWebアプリの開発において広く利用されている**Redux**について、Reduxが担う課題設定と解法のアプローチを解説し、iOSアプリ開発への適用と活用方法を紹介します。ReduxはFluxアーキテクチャのアイデアに大きな影響を受けています。本書の第4章「アーキテクチャのパターンを鳥瞰する」および第8章「Flux」の解説も参照ください。第15章「Reduxの導入例 - 大規模アプリケーションにReduxを導入する」ではReduxを活用してiOSアプリを開発する具体的な手ほどきを解説しています。こちらもあわせて参照ください。

9.1 なぜReduxなのか

9.1.1 背景

Facebook[注1)]は、2014年5月のF8のセッション[注2)]にてFluxアーキテクチャ[注3)]を提唱し、Webアプリ向けのFluxアーキテクチャのリファレンス実装としてFlux[注4)]を公開しました。FacebookはFluxアーキテクチャを開発した背景について、巨大なコードベースでかつ大きな開発組織でもスケールするアーキテクチャを目指していると語っています。

Fluxアーキテクチャの登場以降、Fluxにインスパイアされた多様な実装が多くの開発者によって開発されました。その中でもDan Abramov氏[注5)]によって、2015年8月にリリースされたRedux[注6)]が大きな注目を集めています。ReduxはFluxアーキテクチャのアイデアと、関数型言語のElm[注7)]による複雑性に対するアプローチの影響を大きく受けて誕生しました。

ElmはEvan Czaplicki氏[注8)]によって開発された高信頼性なWebアプリ開発のための言語およびアーキテクチャです。関数型言語であるHaskell[注9)]の影響を受けており、良く似た文法をもっています。Haskellと同様に、副作用を暗黙に生じさせない純粋関数を活用するアプローチを採用しており、コンパイラは厳格な静的型付けのもとで強力な型推論と型検査を行ってくれます。

注 1） http://www.facebook.com
注 2） Rethinking Web App Development at Facebook - Facebook F8 Conference 2014 https://www.youtube.com/embed/nYkdrAPrdcw
注 3） https://facebook.github.io/flux
注 4） https://github.com/facebook/flux
注 5） https://github.com/gaearon
注 6） https://github.com/reduxjs/redux
注 7） http://elm-lang.org
注 8） https://twitter.com/czaplic
注 9） https://www.haskell.org

9.1.2 目的

FluxとElmから影響を受けたReduxはどのような目的のために生まれたのでしょうか。JavaScriptによるWebアプリの開発もモバイルアプリ開発と同様に、高度なユーザー体験の要求に合わせてアプリケーションの複雑性が高まっている背景がありました。

複雑性に起因してアプリケーションの状態がいつ、どうして、どのように更新されるのかをコントロールし把握・理解することは困難になってきています。コードはバグの修正、機能追加による改修やリファクタリングの際に壊れやすく、振る舞いや意図を読み取りにくくなり、アプリケーションの挙動は予測不能になります。

Dan Abramov氏はこの複雑性の根源を**変化（mutation）と非同期性（asynchronicity）を混交することにある**と考え、この課題を解決するために、次の特徴をもつReduxを提唱[注10)]しました。

- **Fluxアーキテクチャの情報の伝播を1方向に制限する特徴を踏襲し、いつどのように更新が起きるかを明瞭にする**
- **Elmアーキテクチャの純粋関数による副作用の排除や、イミュータブルな状態表現の制約を踏襲し、厳格で整合性のとれた状態管理を実現する**

これらの特徴によって、Reduxは**状態変化を予測可能にしよう**と試みています。

リーダビリティ（読みやすさ）

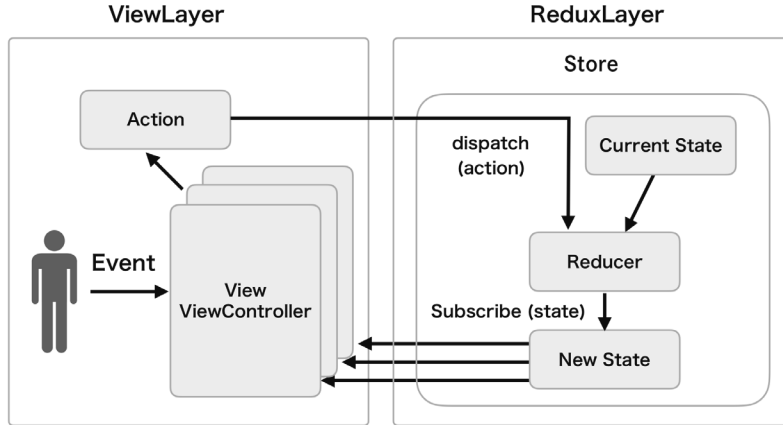
予測可能な形でコードを構造化することを目的とする背景は、コードを書くコストよりも読むコストを下げることに重きが置かれている点にあります。この観点はFacebookの開発思想でもありFluxアーキテクチャからの影響を受けている点でもあります。改善のために幾度となくコードに対して変更が加えられ続ける昨今のサービスでは、他者や将来の自分が記述されたコードから複雑な振る舞いや意図、影響範囲を理解できることが正しいコードの改修への道標となります。

読みやすいコードはレビューがしやすく、改修や不具合の修正時に変更箇所以外のところで不具合が発生するデグレーションの軽減にも貢献します。記述量が増えたとしても大規模なサービスや開発体制にかかわらず、高い品質とスピード感のある開発との両立を実現するためにとても大事な観点です。

注10) https://redux.js.org/introduction/motivation

9.2 Reduxとは

9.2.1 アーキテクチャ概要



● 図 9.1　Redux 概要図

Reduxのアーキテクチャ概要図9.1をもとに、登場する役割と用語について概要を紹介します。解説にあたり、Reduxには存在しない概念ですが便宜的な理由でレイヤーを分割し、呼称を与えました。図左側を画面表示やユーザー操作のハンドリングを担うViewレイヤー、図右側をビジネスロジックを担うReduxレイヤーと命名します。

- **Action**：Reduxレイヤーに対して任意のビジネスロジックの実行や状態の変更を依頼するためのメッセージです。値の太枠内の部分を担いReSwiftオブジェクトで表現します
- **State**：アプリケーションの状態を表現するデータの集合です
- **Reducer**：Actionと現在のStateを入力にとり、新しいStateを出力する関数です
- **Store**：StateとReducerを保持するアプリケーションで単一のインスタンスです。Actionのディスパッチと Reducerの実行、Viewレイヤーからの Stateの購読の機能を有しています

各役割の詳細や、ここで紹介しきれなかった役割については「9.4　ReSwiftとは」にて取りあげます。

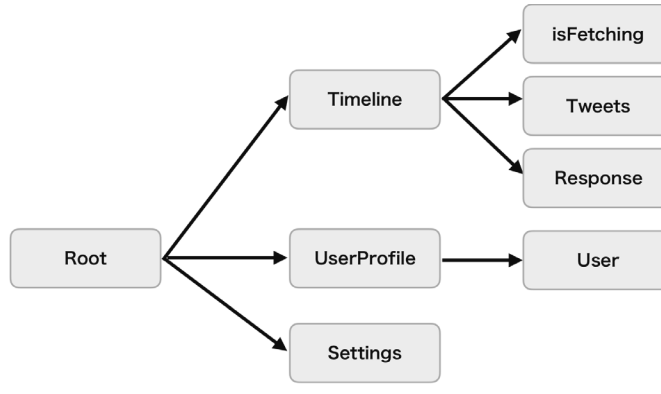
9.2.2　3つの原則

Reduxは**予測可能な形でコードを構造化する**ための骨子として**3つの原則**を提唱しています。Reduxの3つの原則（Redux Three Principles）とは次のとおりです。

- **信頼できる一意となる状態を唯一とする**（Single source of truth）
- **状態はイミュータブルで表現する**（State is read-only）
- **状態の変更は純粋関数で記述する**（Changes are made with pure functions）

Reduxは何かのロジックや振る舞いを部分的に肩代わりしてくれる機能的なもの、いわゆるフレームワークやライブラリではありません。信頼できるこれらの原則にもとづき制約を課して構造化することで、アプリケーションにおいて何が起こっているのか、わかりづらかった変化を明確に理解できるように手助けします。

信頼できる一意となる状態を唯一とする



● 図 9.2　State のオブジェクトツリー

アプリケーションの状態はとても複雑で多義にわたることもあります。その場合、これらの多様な状態は各画面単位で管理したり部分的にシングルトンで管理したりと、状態のインスタンスの参照と管理を適材適所に分散してアプリケーションを構築する場合が多いのではないでしょうか。

Reduxではこのような各状態のインスタンスがあちこちに分散することなく、アプリケーション全体の状態を単一のオブジェクトツリーで管理します。この状態のオブジェクトツリー（図9.2）を**State**と呼びます。Stateは関数を所有しないデータのみで表現されるシンプルなオブジェクトで構成されます。

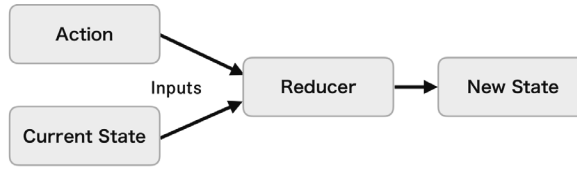
状態を単一のオブジェクトツリーで構成することで、アプリケーションの開発時のデバッグが容易になる恩恵もあります。複雑な状態変化や振る舞いが伴うアプリケーションでは、実行時の状態

が観察しやすいことは挙動の把握やバグの検証において重要な観点です。

Webアプリ開発ではReduxのStateを実行時にモニタリングできる開発支援ツールがそろっており生産性を高めてくれます。

State はイミュータブルで表現する

「Stateをイミュータブルで表現する」とは、作成されたStateが値を変えることのできない不変なインスタンスであることを意味しています。Reducerにより新たなStateが生成されるまでの間、Viewレイヤーで参照している現在のStateはまったく変更されないことが保証されます。よって、イミュータブルな現在のStateを参照している間は、アプリケーション全体で一意で整合性のとれた状態のもとでViewレイヤーの処理が行なえます。



● 図 9.3　State の変更手続き

では、どのようにしてStateの変更を行えばよいのでしょうか。ReduxではStateの変更は**Action**がディスパッチを介して**Reducer**のみ実施できるように制約されています。図9.3ではReducerによるStateの変更手続きを示しています。

Reducerは現在のStateとディスパッチされたActionの２つを入力に受け、新しいStateを出力する関数です。現在のStateはイミュータブルであるため値の変更を行わず、現在のStateのコピーを作ります。Reducerに記述されたビジネスロジックの実行結果をコピーしたStateに適用し、新たなStateとして出力します。このようにReducerではStateの変更を実施しています。

State の変更は純粋関数で記述する

新たなStateの作成を担うReducerは**関数**として表現します。ここでいう関数とは、オブジェクト指向なクラスやメソッドによる記述ではなく、Reducer自身が関数（Reducer関数）として記述されます。さらに、その関数は**純粋関数**であることが求められます。

第 10 章

Clean Architecture

松館 大輝／@d_date, 加藤 寛人／@hkato193

本章では、まずUncle BobことRobert C. Martin氏が提唱したClean Architectureとは何かを紐解き、iOSアプリにおける実現性を考えます。

本章での説明は第1部の内容を前提にしています。まだ読まれていない方はご一読の上で本章を読んでいただくことをお勧めします。

10.1 Clean Architectureとは

Clean Architectureは、Uncle Bobが2012年に彼のブログ[注1) 注2)]で提唱したアーキテクチャのパターンです。

本書では、これまでMVC、MVP、MVVMといったUIとModelとを分離することにフォーカスしたアーキテクチャを説明してきました。これらを**GUIアーキテクチャ**と呼びます（第4章参照）。それに対し、Clean Architectureは**システムアーキテクチャ**に属するアーキテクチャです。UIだけでなくアプリケーション[注3)]全体、つまりModelの内部表現にまで踏み込んだアーキテクチャのパターンです。

あるシステムの1機能を実現するアプリケーションを考えるとき、Clean Architectureはその実現する機能の領域（ドメイン）と技術の詳細に注目し、アプリケーションを4つのコンポーネントに分けます。

- **Entity**：アプリケーションに依存しない、ドメインに関するデータ構造やビジネスロジック
- **Use Case**：アプリケーションで固有なロジック
- **インターフェイスアダプター**：Use Case・フレームワークとドライバで使われるデータ構造を互いに変換する
- **フレームワークとドライバ**：データベース（DB）、Webなどのフレームワークやツールの「詳細」

そして4つが同心円状になるよう、もっとも純粋で他に依存のないEntityを中心に据え、その外にUse Caseを置きます。逆にデータベース／Web／フレームワーク／OSのような、移植や技術遷移で変わりやすいものは最外周に配置します。残るインターフェイスアダプターは内外の変換層として、Use Caseと最外層との間に挟み込んだ階層構造を作ります。

注1） https://8thlight.com/blog/uncle-bob/2012/10/13/the-clean-architecture.html
注2） https://8thlight.com/blog/uncle-bob/2011/11/22/Clean-Architecture.html
注3） 本章で「アプリケーション」と記すときは、iOSアプリに限らない一般的なソフトウェアを指すときに使います。

そして、依存の方向を外から内への一方向に厳密に定めます。

この構造を維持してアプリケーションを作ることで、変わりやすい部分を変えやすく、維持しておきたい部分はそのままにしやすくできます。また、内側にあるEntityやUse Caseは外側のWeb APIサーバーやデバイスドライバなどに依存していないので、それらの完成を待つことなくロジックをテストできます。これがClean Architectureの特徴です。

同じシステムアーキテクチャであるレイヤードアーキテクチャ（第4章参照）と比較すると、レイヤードアーキテクチャではデータベースなどの永続化を担う層がドメインを担当する層の下にいて、ドメインが永続化層に依存しています。一方、Clean Architectureはデータベースを円の外側に置き、依存の方向が逆転しています。さらにデータベースとUse Caseの間には変換層を挟んでいて、切替が容易なしくみをもっている点が異なります。

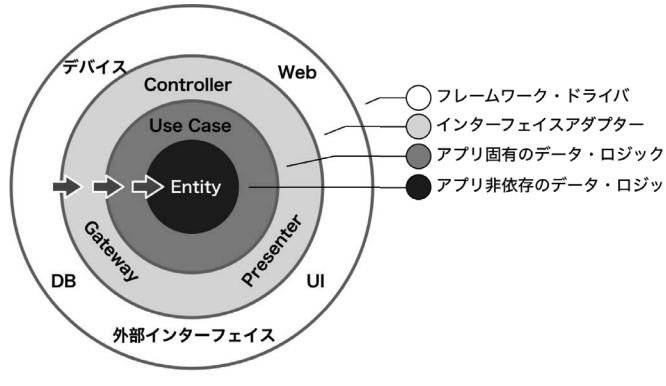
現在のiOSアプリは、UIと単純な処理だけでは成り立たず、Web APIによるデータ通信、データベースへのデータ読み書きなど、外部との連携機能をもつことが当然となっています。また、アプリケーションの要件は複雑・大規模化していて、GUIアーキテクチャが「Model」とひとくくりにするコンポーネントも、内部は加速度的に複雑になっています。さらにアプリはiOSだけでなくAndroidでも同時に開発することも一般的です。ビジネスロジックなどOS非依存の部分を同じように書ければ保守性も高まります。

議論の余地は多分にありますが、Clean Architectureはこうしたシステム全体での大きな共通部分が存在していて、なおかつ複数のプラットフォームに展開するときに効果を発揮する設計パターンといえるでしょう。

本章で、このアーキテクチャを掘り下げて解説します。

10.1.1 依存関係のルール

Clean Architectureの構成要素とその関係は、図10.1の有名な図によって表されます。



● 図10.1　Clean Architecture

Clean Architectureは、冒頭に記したとおりシステムを円状に大きく４つのレイヤーに分けて表現します[注4)]。

1. Entity
2. Use Case
3. インターフェイスアダプター
4. フレームワークとドライバ

円（レイヤー）の内側に行くほど抽象度の高いレイヤーになります。もっとも内側の円こそが、アプリケーションという実現手段からも切り離されたシステムであり、外部の変化から守るべき機能です。

それぞれの構成要素を順に見ていきます。

Entity

Entityは処理の方法に依存しないビジネスロジックであり、データ構造やメソッドの集合体です。外側の層には依存しないため、Use Caseや他の層によってどのように使われるかを気にしません。

Entityが処理の方法に依存しないという点について、Uncle BobはEntityとその他を区別するたとえを次のように記しています[注5)]。

> たとえば、銀行がローンにN%の利子を付けているとすると、それは銀行のお金を生むためのビジネスルールになる。利子をコンピュータで計算しようと、そろばんで計算しようと、全く関係はない。

このたとえでは、利子を計算するルール（ビジネスロジック）はEntityの一員で、コンピュータだろうとそろばんだろうと同じように計算されるべき対象です。計算役がいなくても存在するルールです。つまり境界は利子の計算ルールと計算役のコンピュータ／そろばんとの間に存在し、後者はあくまでアプリケーション固有の概念か実装の詳細です。Entityには外部変化による影響がないものだけが存在することになります。

Use Case

Use Caseは、Entityを使ってアプリケーション固有のビジネスロジックを実現します。アプリケーション固有のビジネスロジックとは、構築対象のアプリケーションに対してのみ有効な処理であるということです。Entityが複数のアプリケーションで共有できることを念頭に置いているのとは対照的です。

複数のアプリケーションを作るわけではないので、EntityとUse Caseとを区別する必要はないのでは、と思う方もいるかもしれません。しかし、ここでいう複数のアプリケーションとは、並行で作

注４）　４つ円があることが重要なのではなく、依存関係のルールが常に存在していることが重要です。円の数は４より多くても問題ありません。
注５）『クリーンアーキテクチャ 達人に学ぶソフトウェアの構造と設計』p.189 より引用

るアプリケーションのことのみを想定しているわけではありません。将来的に作り変える可能性のある機能や要件のことも見据えています。

例として、あるコンテンツを参照するだけのアプリケーションと、参照・書換を行う管理用アプリケーションの２つを作る場合を考えます。この場合、コンテンツはEntityに配置しますが、参照処理と参照・書換処理の２つはUse Caseに配置し、コンテンツにアクセスする処理を記述します。

また、Use Case層にはUIに関する処理は書きません。入出力のための出入口（ポート）は存在しますが、そのポートにどのような経路から入力があって、どこへ出力するのかは知りません。それらは次に説明するインターフェイスアダプター／フレームワークとドライバ層に書きます。

インターフェイスアダプター

インターフェイスアダプターは、円の内外に合わせてデータやイベントを変換するためのレイヤーです。Use CaseやEntityで扱っているデータ表現をSQLやUI用のデータに変換したり、逆にデータベースやWebからのデータをUse CaseやEntityで使われる表現に変換するなど、両縁のためにつなぎの役割をこなします。いわゆるPresenterやControllerがこの層に属します。

また、インターフェイスアダプターはUse Caseと最外層とを接続する役割を担うことから、Use Caseの入出力ポートを外層の何に接続するかを決定する責務も持ちます。

フレームワークとドライバ

UI、データベース、デバイスドライバ、Web APIクライアントなどがこの最外層に該当します。どれも実装の詳細で、環境や顧客の要求変化にもっとも影響を受ける場所です。

この層には特定の条件下でのみ有効なコードがここに集まります。たとえば、FlutterやReact Nativeでのネイティブコードはこのレイヤーに配置します。

特筆すべきは、UIや実装先OSの種類、さらにはフレームワークといった環境すらもこのレイヤーで扱われることです。UIKitやAlamofireもこのレイヤーです。どれもビジネスロジックとは無関係であり、状況により実現手段がよく変化するためです。

いささか厳密すぎるように感じるかもしれませんが、このルールにより、円の内部は外界から完全に切り離され、クリーンさを維持できるというわけです[注6)]。

この層は実装の詳細であるため、データベースやOSのようにもともと大量のコードが存在することもあります。しかし、それらは円とは独立したコンポーネントであり、円の内側とは本来関係がないものです。したがって、データベースなどのソースコードを単に置くだけでなく、インターフェイスアダプターと通信するための「つなぎ」となるコードも、この層に書きます。

注 6）極端な例を挙げれば、タイマーなども最外層です。

Column.　誰がアーキテクチャを組み立てるのか

Clean Architectureの各レイヤーを組み立てて円構造に仕立て上げるのは、**Mainコンポーネント**と呼ばれる、アプリケーション起動時のセットアップを担うコンポーネントの役割です。このコンポーネントはフレームワークとドライバ層と同じ最外層に置かれ、起動後ただちにそれぞれのクラスをインスタンス化してアーキテクチャを組み上げます。

このモジュールはすべてのクラスを知っているため、アプリケーションの中でもっとも汚れ役を担うことになります。もちろん、Factoryパターンを使ってMainコンポーネントから具体的なクラスを隠蔽するのも問題ありません。

iOSアプリにおいては、Application Delegateか、あるいはそこから呼ばれる専用オブジェクトが、Mainコンポーネントとして動くことになります。

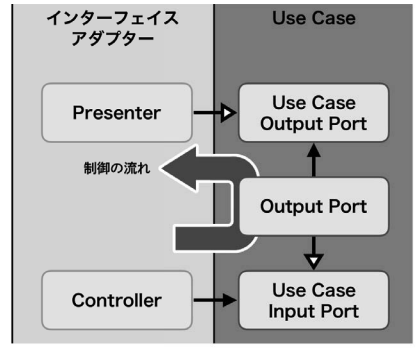
10.1.2　レイヤー間の通信

レイヤー間の相互関係には重要なルールがあります。それは内側の円は外側の円からのみ参照されることです。内側の円が、外側の円を直接参照することはありません。言い換えると、内側のクラスが外側のクラスや関数を直接参照することがあってはいけないのです。なにか別の、間接的な方法で内側から外側への通信を実現しなければいけません。

そのルールをふまえた上で、レイヤー間の相互通信がどのように行われるかを見てみましょう。

依存関係逆転の原則

図10.2は、Clean Architectureの２つの層を取り出し、その間の通信がどのような関係の下で行われるかを示したものです。



● 図 10.2　境界を跨いだ制御の流れ

第 11 章

アプリの起動経路
- Application Coordinatorの導入

松館 大輝／ @d_date

本章ではアプリ内の画面遷移処理のパターンとしてCoordinatorを紹介し、起動経路を整理するための指標を示します。

11.1 Coordinator

Coordinatorパターンは、Soroush Khanlou氏によってiOSアプリに持ち込まれました。2015年のNSSpainでのトーク[注1)]と、ブログ[注2)]で内容を読めます。

画面遷移の処理は、一般的にView Controllerが受けもちます。画面遷移をしたい場合にはView Controller内で次のView Controllerをインスタンス化し、Navigation ControllerへのpushやModalのpresentを行うのが一般的です。

● リスト 11.1　通常の画面遷移

```
extension ViewController: UICollectionViewDelegate {
    func collectionView(_ collectionView: UICollectionView,
                        didSelectItemAt indexPath: IndexPath) {
        let object = objects[indexPath.item]
        let nextVC = NextViewController(object: object)
        navigationController?.pushViewController(nextVC, animated: true)
    }
}
```

しかし裏を返すと、View Controllerは次のView Controllerのことを知っていることになります。遷移元のView Controllerと遷移先のView Controllerの関係が 1 対 1 である場合には大した問題にはなりません。ですがそのView Controllerを使い回したり、遷移先が複数存在するような場合、遷移先が特定のView Controllerに依存していることで、遷移ロジックが肥大化します。

これを解決するために、Soroush氏はView Controllerの上位レイヤーとして、画面遷移を管理する**Coordinator**を導入することを提案しました。

注 1)　Soroush Khanlou (2015)『Coordinators』https://www.slideshare.net/secret/3jJlEE1weo0RRl
注 2)　Soroush Khanlou (2015)『The Coordinator』http://khanlou.com/2015/01/the-coordinator/

11.2 Application Coordinator

Coordinatorを導入する場合、アプリ全体を管轄する責務を持ったCoordinatorが1つ必要です。これをApplication Coordinatorと呼びます。Application CoordinatorはAppDelegateが所有し、ルートビューに対するCoordinatorとなります。

Application Coordinatorをルートとして、1つのView Controllerにつき1つのCoordinatorが存在します。そして画面遷移の経路に沿った親子関係を構築します。たとえばをTab Barをルートとするアプリでは、それぞれのTab Bar Itemで、対応するNavigation Controllerごとに1つのCoordinatorが存在し、それをタブ全体のCoordinatorが親として持つことになります。

Application CoordinatorはApplication Controllerとも呼ばれていて、Martin Fowler氏のエンタープライズアーキテクチャパターンに登場[注3)]する設計パターンのひとつです。

11.2.1 Application Coordinatorの実装

さっそくApplication Coordinatorを作ってみましょう。

まずは`Coordinator`をprotocolとして定義します。

● リスト11.2 Coordinator

```
protocol Coordinator {
    func start()
}
```

ここでは`start()`というメソッドのみを定義します。次にこの`Coordinator`に準拠した`AppCoordinator`を実装します。

● リスト11.3 AppCoordinator

```
final class AppCoordinator: Coordinator {
    private let window: UIWindow
    private let rootViewController: UITabBarController
    private var repoListCoordinator: RepoListCoordinator

    init(window: UIWindow) {
        self.window = window
        rootViewController = .init()
```

注3) https://martinfowler.com/eaaCatalog/applicationController.html ルーツはこの論文（https://ieeexplore.ieee.org/document/991332）に端を発するようですが、本書では詳細を割愛します。

```
        let repoNavigationController = UINavigationController()
        self.repoListCoordinator = RepoListCoordinator(
          navigator: repoNavigationController)
        rootViewController.viewControllers = [repoNavigationController]
    }
    func start() {
        window.rootViewController = rootViewController
        repoListCoordinator.start()
        window.makeKeyAndVisible()
    }
}
```

AppCoordinatorのイニシャライザはUIWindowを引数に取ります。AppDelegateのdidFinishLaunchingWithOptionsでAppCoordinatorを初期化します。RepoListCoordinatorは、最初に表示されるView Controllerと対応するCoordinatorです。次の項で実装を解説します。

AppCoordinatorの初期化は次のとおりです。

● リスト11.4　AppCoordinatorの初期化

```
final class AppDelegate: UIResponder, UIApplicationDelegate {
    var window: UIWindow?
    private var appCoordinator: AppCoordinator?

    func application(_ application: UIApplication,
                     didFinishLaunchingWithOptions
        launchOptions: [UIApplication.LaunchOptionsKey : Any]? = nil) -> Bool {

        let window = UIWindow(frame: UIScreen.main.bounds)
        self.window = window

        let appCoordinator = AppCoordinator(window: window)
        appCoordinator.start()
        self.appCoordinator = appCoordinator
        return true
    }
}
```

application(_:didFinishLaunchingWithOptions:)でwindowとAppCoordinatorを初期化しています。初期化したあとは、Coordinatorのstart()を呼びます。start()では、最初に表示される画面に対応するCoordinatorのstart()をコールしています。これにより画面遷移が始まり、この時点でAppDelegateは遷移先のView Controllerを知らずにすむようになりました。

11.3 Coordinatorによる画面遷移

さて、実際Coordinatorによって画面遷移はどのように変わるのでしょうか。Coordinatorによって View Controllerの遷移処理が隠蔽されている様子を実際にみてみます。

起動するとTable Viewが表示されて、セルをタップすると詳細画面に遷移するようなアプリをサンプルとします。

なお紙面の都合上、Coordinatorの理解に重要ではない箇所は極力省略しています。お手元にサンプルコードを用意してあわせてご覧ください。

11.3.1 RepoListViewController

まずは`RepoListViewController`という画面を作成します。

● リスト 11.5 RepoListViewController

```
protocol RepoListViewControllerDelegate: AnyObject {
    func repoListViewControllerDidSelectRepo(_ repo: GitHubRepoModel)
}
final class RepoListViewController: UIViewController {
    weak var delegate: RepoListViewControllerDelegate?
    var repoList: [GitHubRepoModel] = []
    ...
}
```

`RepoListViewControllerDelegate`はこのあと使うので先に実装しておきます。コードのみでレイアウトしたTable Viewを載せているView Controllerです。

11.3.2 RepoListCoordinator

次にRepoListViewControllerに対応するCoordinatorを作成します。

● リスト11.6 RepoListCoordinator

```
final class RepoListCoordinator: Coordinator {
    private let navigator: UINavigationController
    private var repoListViewController: RepoListViewController?

    init(navigator: UINavigationController) {
        self.navigator = navigator
    }
    func start() {
        let viewController = RepoListViewController()
        viewController.delegate = self
        navigator.pushViewController(viewController, animated: true)
        self.repoListViewController = viewController
    }
}
```

RepoListCoordinatorの初期化には、Navigation Controllerを必要とします。Coordinatorが Navigation Controllerを保持することで実際の遷移処理を受けもちます。start()がコールされることで、RepoListCoordinatorに対応するView ControllerであるRepoListViewControllerが表示されます。

11.3.3 詳細画面への遷移

セルをタップしたら詳細画面に遷移するようにします。tableView:didSelectRowAt:indexPathを見てみましょう。

● リスト11.7 セルが選択されたときに画面遷移する

```
extension RepoListViewController: UITableViewDelegate {
    func tableView(_ tableView: UITableView,
                   didSelectRowAt indexPath: IndexPath) {
        let repo = repoList[indexPath.row]
        delegate?.repoListViewControllerDidSelectRepo(repo)
        tableView.deselectRow(at: indexPath, animated: true)
    }
}
```

ここでのポイントは、遷移処理をRepoListViewControllerDelegateに任せていて、Navigation Controllerが書かれていないことです。このRepoListViewControllerDelegateはCoordinatorが保持しています。

第 12 章

画面遷移のパターン - Routerの導入

田中 賢治／@ktanaka117

iOSアプリにおける**Router**の役割は、**遷移先の画面を生成し遷移処理の責務を担うこと**です。Routerを導入することで、複雑になりがちなView Controllerから画面の生成と遷移の処理を引きはがし、遷移に関する責務の分割を行えます。

Routerという言葉はMVCやClean Architectureなどのアーキテクチャパターンを指す用語ではなく、他のアーキテクチャパターンと組み合わせて使うコンポーネントのことを指します。MVCにおけるControllerや、Clean ArchitectureにおけるUse Caseの役割のような単位で存在します。

12.1 Routerを導入するモチベーション

`UIViewController`クラスは、画面やユーザーからのインプットとその応答など管理対象が多く、もともと責務が肥大化しがちなクラスです。責務の肥大化はバグの原因となるコードの複雑化と、可読性の低下を招きます。Routerは画面遷移の処理をひとつの責務ととらえて、単一責任原則にもとづき各画面の処理から横断的に切り出します。

Routerを導入するモチベーションは次のとおりです。

- **単一責任原則に則って、肥大化するView Controllerから画面の生成と遷移処理を実施する責務を引きはがしたい**
- **遷移先が多い画面で遷移処理を一箇所にまとめて、簡潔に管理しやすくしたい**

先述したモチベーションを実現するための、Routerの役割は次のとおりです。

- **遷移先のView Controllerの生成**
- **遷移先のView Controllerが依存するコラボレーターの生成と注入**
- **画面遷移の実施方法の定義**

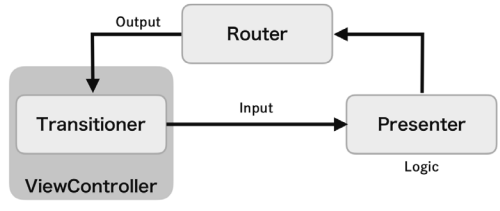
そこで次節では既存の`UIViewController`クラスを活かしつつ、Swiftらしく書くための具体例を紹介します。しかしUIKitにおける画面遷移の処理は`UIViewController`上に実装されていて、完全な切り離しは不可能です。

12.2 Routerの実装例

Routerの実装方法はいくつかありますが、本書ではMVPをベースにしたRouterの実装例を紹介します。MVPについては第6章「MVP」を参照ください。

Routerを実装する手順は次の3ステップです。

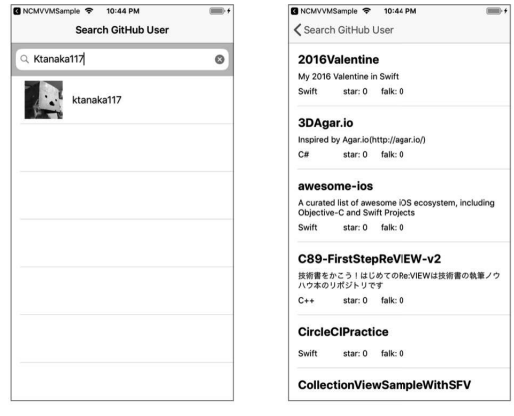
1. View Controllerから画面遷移の責務を切り離す
2. 画面遷移の責務を担うRouterを作成する
3. PresenterからRouterに画面遷移を指示する



● 図 12.1　本書で紹介する Router の図

説明に使うコードは本書のサンプルコードに含まれています。ぜひ本書で解説している内容と比べながらコードリーディングしてみてください。

今回解説に使うのは、GitHubのユーザー検索を実施し、検索結果のユーザーのRepository一覧を表示する簡単なアプリです。図12.2がサンプルアプリのスクリーンショットです。



ユーザー検索　　リポジトリー覧

● 図 12.2　Router のサンプルアプリ

12.2.1 View Controller から画面遷移の責務を切り離す

View Controllerから画面遷移処理を切り出して、Presenterが画面遷移処理をRouterに指示できるようにするには、まずUIViewController（UINavigationController）クラスが持つ画面遷移のメソッドを抽象化する必要があります。

そこでprotocolとしてTransitionerを定義し、その中に画面遷移のメソッドをひととおり宣言します。

```
protocol Transitioner: AnyObject where Self: UIViewController {
    func pushViewController(_ viewController: UIViewController, animated: Bool)
    func popViewController(animated: Bool)
    func popToRootViewController(animated: Bool)
    func popToViewController(_ viewController: UIViewController, animated: Bool)
    func present(viewController: UIViewController,
                 animated: Bool,
                 completion: (() -> ())?)
    func dismiss(animated: Bool)
}
```

この抽象化は元からあるUIViewController（UINavigationController）クラスのインターフェイスとまったく同じもので構いません。具体的な遷移先View Controllerを設定するのは、あとで登場するRouterが担当します。

Transitionerで宣言したメソッドから、UIViewController（UINavigationController）の実装を呼び出せるよう、protocol extensionを使った共通処理として定義します。

```
protocol Transitioner: AnyObject where Self: UIViewController {
    func pushViewController(_ viewController: UIViewController, animated: Bool)
    func popViewController(animated: Bool)
    func popToRootViewController(animated: Bool)
    func popToViewController(_ viewController: UIViewController, animated: Bool)
    func present(viewController: UIViewController,
                 animated: Bool,
                 completion: (() -> ())?)
    func dismiss(animated: Bool)
}
extension Transitioner {
    func pushViewController(_ viewController: UIViewController,
                            animated: Bool) {
        guard let nc = navigationController else { return }
        nc.pushViewController(viewController, animated: animated)
    }
    ...
    func present(viewController: UIViewController,
                 animated: Bool,
```

```
                completion: (() -> ())? = nil) {
        present(viewController, animated: animated, completion: completion)
    }
    func dismiss(animated: Bool) {
        dismiss(animated: animated)
    }
}
```

Transitionerの宣言にある「where Self: UIViewController」の記述によって、このprotocolに準拠するクラスをUIViewControllerに限定するところがポイントです。これによってprotocol extension内で実際の遷移処理を記述できるようになります。

そしてUIViewControllerのサブクラスをTransitionerに準拠させます。MVPの場合はPresenterがViewへの出力を管理するため、Viewをprotocolとして抽象化していることが多くあります。その場合、次のようにView Controllerが準拠する出力用のprotocolに対してTransitionerを準拠させます。

```
protocol SearchUserViewProtocol: AnyObject,
                                 Transitioner where Self: UIViewController {}
extension SearchUserViewController: SearchUserViewProtocol {}
```

これによって、Presenterが管理するRouterから、protocol extensionに定義した共通の画面遷移処理を呼び出せるようになります。

以上で画面遷移の責務をView Controllerから切り離す実装は完了です。

12.2.2 画面遷移の責務を担うRouterを作成する

次に具体的な画面遷移処理を管理するRouterについて解説します。

Routerの基本構造は次のようになります。

```
protocol SearchUserRouterProtocol: AnyObject {
    func transitionToUserDetail(userName: String)
}
final class SearchUserRouter: SearchUserRouterProtocol {
    private(set) weak var view: SearchUserViewProtocol!

    init(view: SearchUserViewProtocol) {
        self.view = view
    }
    func transitionToUserDetail(userName: String) {
        // ここでTransitionerのメソッドを使って画面遷移
    }
}
```

protocolで遷移処理を関数として宣言していき、Presenterから扱いやすくします。画面遷移の管理対象であるviewを弱参照で保持することで、そのviewに対する画面遷移処理と遷移先のView Controllerとを整備します。

ひとつ具体的なRouterの実装を見てみましょう。ユーザー詳細画面へ遷移するtransitionToUserDetail(userName:)を実装したのが、次のコードです。

```
protocol SearchUserRouterProtocol: AnyObject {
    func transitionToUserDetail(userName: String)
}
final class SearchUserRouter: SearchUserRouterProtocol {
    private(set) weak var view: SearchUserViewProtocol!

    init(view: SearchUserViewProtocol) {
        self.view = view
    }
    func transitionToUserDetail(userName: String) {
        let userDetailVC = UIStoryboard(name: "UserDetail", bundle: nil)
            .instantiateInitialViewController() as! UserDetailViewController
        let model = UserDetailModel(userName: userName)
        let router = UserDetailRouter(view: userDetailVC)
        let presenter = UserDetailPresenter(
            userName: userName,
            view: userDetailVC,
            model: model,
            router: router)
        userDetailVC.inject(presenter: presenter)

        view.pushViewController(userDetailVC, animated: true)
    }
}
```

SearchUserRouterで遷移先のView Controllerの生成と、その依存を解決しています。遷移先のView Controllerに必要な要素は画面遷移関数の引数として宣言し、呼び出し側（＝遷移元）から渡します。画面遷移を行っているview.push(userDetailVC, animated: true)では、先ほど宣言してきたTransitionerを利用しています。

これでView Controllerから切り離した画面遷移の責務をRouterへ移す実装は完了です。

第 13 章

第2部まとめ - アーキテクチャの選定基準

松館 大輝／ @d_date

本章は第２部のまとめとして、これまでに紹介したアーキテクチャパターンをどのような基準で選ぶべきか、私たちからの選定ガイドを紹介します。

ただし先に断っておくと、アーキテクチャ選定のパラメータは無数にあり、ここで紹介する範囲だけで厳密に選べるようなものではありません。あくまで１つの視点ということで、肩の力を抜いてお読みください。

1. アプリに求められる機能要件は何か
2. アプリに求められる非機能要件は何か
3. そのアーキテクチャパターンに精通しており、リードできる開発者がいるか
4. チームの人数やスキルセットに合っているか
5. アーキテクチャパターンが目的になっていないか
6. そのアーキテクチャパターンが好きになれるかどうか

13.1 アプリに求められる機能要件は何か

アプリの複雑度によって、適切なアーキテクチャは変わります。

ひとつの指標は、「ドメインロジックが複雑かどうか」です。

View Controllerの肥大化をきっかけにアーキテクチャを考え始めることはよくあります。画面の要素が増えたり、外部要因が増えれば増えるほど、View Controllerが肥大化して見通しが悪くなり、メンテナンスしにくくなります。そうした事態が予見される場合には、MVPやMVVMのように、ドメインロジックとプレゼンテーションロジックを分割しやすいアーキテクチャパターンを採用すると、画面に必要なデータと実際のViewとが分割しやすくなります。逆に、設定画面やチュートリアル画面のような、要素が少なくシンプルな画面にアーキテクチャパターンを無理やり持ち込むことで、工数の肥大化などのオーバーエンジニアリングを招くことがあります。第3章を思い出してください。単一責任原則から見れば、「変更の理由が同じもの」はひとつのモジュールにまとめられているべきなのです。

また、別の指標で「画面間の状態整合性が必要かどうか」も挙げられます。

タブバーで画面を切り替えたときに状態を同期するアプリを考えます。一番左のタブにはお気に

入りをつけられるニュースフィード画面があるとしましょう。別のタブにはお気に入りの一覧画面があります。このとき、ニュースフィード画面のある記事のお気に入りボタンが押されると、お気に入り一覧画面にも要素を追加/削除することになります。このときのお気に入り一覧を表示するためには、パフォーマンスの観点からもクライアント側で状態管理すべきです。WebAPIを呼び最新情報を取得する方法は良策とはいえません。そうした場合には、Reduxのような画面間の状態整合性を考慮したアーキテクチャパターンを選定しておくことで楽に実装できる場合があります。

13.2 アプリに求められる非機能要件は何か

アプリのアーキテクチャパターンを選ぶとき、それが新規開発なのか、ボトルネック解消のためのアーキテクチャの変更なのか、さまざまな状況があります。こうした状況において「こういう機能を持っている」「〜ができる」といった機能要件はイメージしやすく、スペックシートに残ることが多いのですが、その背後にある非機能要件については暗黙知のままで進んでしまうことがよくあります。パターンとは「ある問題に対する解決策」です。問題はアプリの機能要件だけでなく、非機能要件にも潜在しています。アーキテクチャパターンを適切に選ぶためには、まずはアプリに求められる非機能要件をきちんと聞き出し、解決すべき問題が何かを特定する必要があります。

非機能要件とは、機能要件以外全般を指しますが、日本情報システムユーザー協会（JUAS）が発行した『非機能要件要求仕様定義ガイドライン』では「機能性」「信頼性」「使用性」「効率性」「保守性」「移植性」「障害抑制性」「効果性」「運用性」「技術要件」という10種類に分類しています。[注1)]

よくあるのは次のような要求でしょう。

- 施策をどんどん打ちたいので、機能追加／削除しやすい方がよい
- 不具合が出ることは影響が甚大になるので、できるだけ不具合が抑えられる方がよい

２つの要求はある程度重なります。第２章でも説明したとおり、適切に設計することは手戻りを防ぎ、早く作ることにつながります。また、新たな機能提供をするために、変わらないドメインロジックをEntity層に閉じ込め、きっちりUse Case層と分ける必要があるという見方もあります。

それでは、なんとしてでもClean Architectureを採用すべきでしょうか。

しかし、次のようなケースにおいてClean Architectureはよくない効果を及ぼします。

- スタートアップでコアドメイン自体がまだ安定しておらず、根本的な概念の見直しが度々発生する
- A/Bテストの都合上、UIの都合を完全に分離したModelの設計が難しい
- クライアントのビジネスロジックが薄い

注1) https://www.juas.or.jp/library/books/445011/ からダウンロードできるExcelで、それぞれの項目の詳細な定義、下位分類も含めて指標が整理されているので確認できます。

このような場合、Model層を厳格に作り込むことは避け、最低限のMVC、MVPなど最低限のGUIアーキテクチャパターンを採用するに留めるべきです。

また、不具合を抑えることについても、ソフトウェアテストの７つの原則には「バグゼロの落とし穴」というものがあります。第２章で説明したのは「チェッキング（動作確認）」のテストであり、「テスティング（品質保証）」のテストについては話が別です。バグには許容度があります。重要でない場合に表示の整合性がおかしくなることを許容し、ある程度の柔軟性を与えて機能追加のしやすさを優先すべき局面は、実際の現場ではよくあります。

上記が必ずしも作るアプリにマッチするわけではありません。しかし、このようにアプリ設計の際には、どのような要求があるのかを明確にすることが重要です。

13.3 そのアーキテクチャパターンに精通しており、リードできる開発者がいるか

どんなアーキテクチャパターンを採用するとしても、そのパターンの経験があったり、パターンに精通していることが望ましいです。そのアーキテクチャパターンで開発した経験がある開発者であれば、なぜそれを採用することにメリットがあるのか、そしてどのようなところがネックになるのかを知っているはずです。間違った解釈で進みかけたときにも道を正してくれます。そのような開発者がいることで、ペアプログラミングやレビューをしながら開発を進められるだけでなく、ネックになる箇所に適切な落としどころをつけながらチーム開発を進められます。

経験のないアーキテクチャパターン導入にはリスクが伴いますが、実際に導入しないとそのアーキテクチャパターンを深く理解することはできませんし、開発者へのモチベーションにも影響します。そうした場合は、まずは小さく始めることを考えてみてください。過去に前例があるアーキテクチャパターンであれば、次のような簡単なことから始めてみましょう。[注2)]

- サンプルコードを読んでみる [注3)]
- 一画面レベルの簡単なアプリを作ってみて、チームメンバーと議論してみる

注 2） 使えそうなアーキテクチャパターンが見つかれば、他チームのメンバーにも布教してみてください。気付いたときには、あなたがチームをリードする存在になっているかもしれません。
注 3） 本書にも各アーキテクチャパターンのサンプルコードを掲載しているので参考にしてください。

第3部

設計をサービスに導入する

第 14 章

Fluxの導入例

鈴木 大貴／@marty_suzuki

第8章ではflux-concepts[注1)]に沿って、Fluxアーキテクチャを iOSアプリに実装する方法について説明しました。本章では実際の運用を想定したiOSアプリに、Fluxアーキテクチャを導入する実践的な実装方法について説明します。

14.1 作成するアプリの全体像

本節では、第8章で実装した**GitHubのリポジトリ検索及びお気に入り追加アプリ**を、実際の運用を想定した状態に修正していきます。

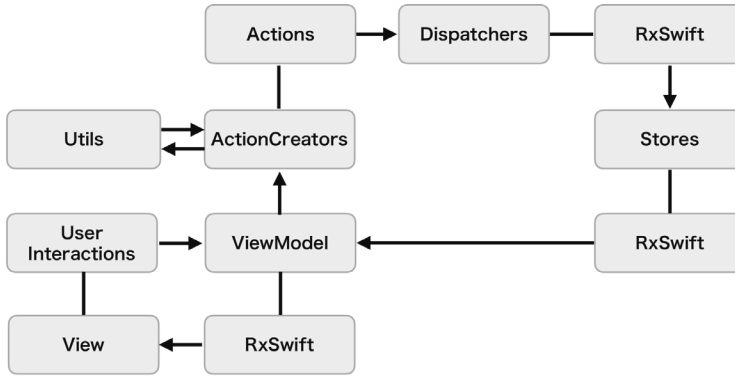
● 図 14.1　GitHub のリポジトリ検索及びお気に入り追加

注 1)　https://github.com/facebook/flux/tree/3.1.1/examples/flux-concepts

このアプリの機能は次の３点です。

- GitHubのリポジトリ検索
- GitHubのリポジトリの詳細を表示
- ローカルにお気に入りのリポジトリを保存し一覧表示

機能自体は第8章で実装したアプリと変わりませんが、図14.2のように全体像が大きく変わります。



● 図 14.2　ViewModel を用いた Flux アーキテクチャの全体像

図14.2で示しているように、**RxSwift**[注2)]と**MVVMアーキテクチャのViewModel**を導入していることが大きく変わります。純粋なFluxアーキテクチャに加え、Viewコンポーネントの肥大化を防ぐためにViewModelを組み合わせて利用します。また純粋なFluxアーキテクチャの次の項目に対してRxSwiftを利用することで、よりデータフローがわかりやすくなります。

- Dispatcherの`register(callback:)`の代替
- Storeの`addListener(callback:)`の代替
- Viewへのバインディング

まずは、`Dispatcher`・`Store`・`ActionCreator`・`View`にRxSwiftを導入します。

注 2）Swift で Reactive Programming を表現でき、非同期処理やイベント処理を宣言的に記述できるライブラリです。 https://github.com/ReactiveX/RxSwift

14.1.1 Dispatcher の実装

Dispatcher でRxSwift を利用した実装について説明します。「8.3.1　リポジトリ検索画面」では Dispatcher.register(callback:) で渡した callback を Dictionary で保持し、Dispatcher.dispatch(_:) で渡された Action を Dictionary で保持している callback に渡すことで Dispatcher の実装を実現していました。それらはRxSwift を利用することで、リスト 14.1 のように実装できます。

● リスト 14.1　Dispatcher

```
final class Dispatcher {
    static let shared = Dispatcher()

    private let _action = PublishRelay<Action>()

    func register(callback: @escaping (Action) -> ()) -> Disposable {
        return _action.subscribe(onNext: callback)
    }
    func dispatch(_ action: Action) {
        _action.accept(action)
    }
}
```

Dispatcher では、_action を持っています。Dispatcher.register(callback:) では、引数で受け取った callback を _action.subscribe(onNext: callback) として、流れてくる Action を監視します。戻り値は Disposable になっているので、Dispatcher.register(callback:) の呼び出し元で監視を解除できます。そのため Dispatcher.unregister() を実装する必要はなくなります。

また Dispatcher.dispatch(_:) では、引数で受け取った Action を _action.accept(action) で流しています。そのため Dispatcher.register(callback:) の callback に Action が渡され、監視先で Action を受け取れます。

14.1.2 Store の実装

RxSwift を利用した Store を実装する方法について説明します。「8.3.1　リポジトリ検索画面」では NotificationCenter を利用して、Store の変更を監視しているオブジェクトに対して変更を通知していました。それらを RxSwift を利用することで、GitHub のリポジトリ検索を担う Store はリスト 14.2 のように実装できます。

● リスト 14.2　SearchRepositoryStore

```
final class SearchRepositoryStore {
    static let shared = SearchRepositoryStore()

    var repositories: [GitHub.Repository] {
        return _repositories.value
    }
    var repositoriesObservable: Observable<[GitHub.Repository]> {
        return _repositories.asObservable()
    }
    private let _repositories = BehaviorRelay<[GitHub.Repository]>(value: [])

    var errorObservable: Observable<Error> {
        return _error.asObservable()
    }

    private let _error = PublishRelay<Error>()

    private let disposeBag = DisposeBag()

    init(dispatcher: Dispatcher = .shared) {
        dispatcher.register(callback: { [weak self] action in
            guard let me = self else { return }
            switch action {
            case let .searchRepositories(repositories):
                me._repositories.accept(me._repositories.value + repositories)
            case .clearSearchRepositories:
                me._repositories.accept([])
            case let .error(error):
                me._error.accept(error)
            }
        }).disposed(by: disposeBag)
    }
}
```

SearchRepositoryStoreではGitHubのリポジトリ一覧を保持するため、_repositoriesを定義します。BehaviorRelayを外部に公開するとsetterも公開することになってしまうため、Storeの外部からBehaviorRelayで保持している値に意図しない更新が可能になってしまいます。

そこで外部から値に直接アクセスするためのrepositoriesと、_repositoriesを購読できるようにするためのrepositoriesObservableを定義します。errorは発生した際に処理をするために必要ですが、状態を保持してまで処理をすることは少ないため _errorを定義し、Dispatcherからerrorが流れてきた場合はそのまま流すだけにします。

Action.searchRepositoriesが流れてきた場合は、現状のリポジトリの配列を取得し渡ってきた

配列と結合した後に、配列をaccept(_:)に渡します。Action.clearSearchRepositoriesが流れてきた場合は、空の配列をaccept(_:)に渡します。Action.errorが流れてきた場合は、そのままerrorをaccept(_:)に渡します。

「8.3.1　リポジトリ検索画面」ではStore.addListener(callback:)によってStoreに更新があったことは監視できていましたが、どの値が更新されたかまではわからない実装でした。しかしRxSwiftを利用することで、Storeで公開されているそれぞれのプロパティをObservableにできるため、プロパティごとに購読できます。

14.1.3　ActionCreatorの実装

ActionCreatorでRxSwiftを利用した実装について説明します。まずは、ActionCreatorで処理するAPI通信で、RxSwiftを利用したprotocolをリスト14.3のように定義します。

● リスト14.3　GitHubApiRequestable

```
protocol GitHubApiRequestable: AnyObject {
    func searchRepositories(query: String,
                            page: Int) -> Observable<([Repository], Pagination)>
}
```

RxSwiftを利用しない場合だとclosureを利用してAPIの通信結果を返していましたが、RxSwiftを利用するとObservableで結果を返せます。リスト14.3のprotocolをActionCreatorで利用したのが、リスト14.4です。

● リスト14.4　ActionCreator

```
final class ActionCreator {
    static let shared = ActionCreator()

    private let dispatcher: Dispatcher
    private let apiSession: GitHubApiRequestable

    init(dispatcher: Dispatcher = .shared,
         apiSession: GitHubApiRequestable = GitHubApiSession.shared) {
        self.dispatcher = dispatcher
        self.apiSession = apiSession
    }
    func searchRepositories(query: String, page: Int = 1) {
        dispatcher.dispatch(.searchQuery(query))
        dispatcher.dispatch(.isRepositoriesFetching(true))
        _ = apiSession.searchRepositories(query: query, page: page)
            .take(1)
```

第 15 章

Reduxの導入例
- 大規模アプリケーションにReduxを導入する

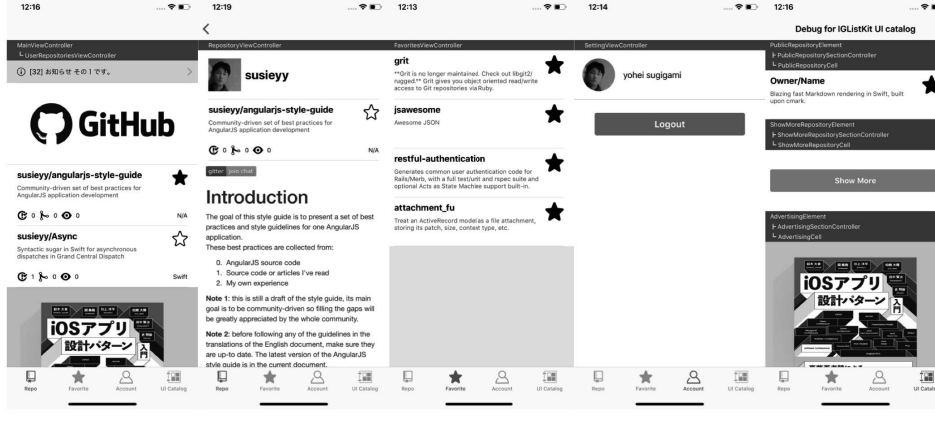
杉上 洋平／@susieyy

株式会社FOLIOの新規アプリ開発にあたりReduxを導入しました。採用の背景はサービスの求める品質要件の高さと、Reduxの不変さや純粋関数による堅牢さが合致したからです。また、次の観点も採用する後押しとなりました。

- 1画面あたりの取りうる状態が多様で、整合性を保って状態変化を管理したかった
- ビジネスロジックの可読性において、Pull Request時にレビュアーが理解しやすく影響範囲も把握できるようにしたかった

本章では開発で得た知見を活かしつつ、さらに改良や試行錯誤を加えた新たなReduxの実装を紹介します。第8章で解説したReSwiftライブラリをRedux実装の中核に添え、さらにRxSwiftライブラリと組み合わせることで、より表現豊かな記述を用いたアプリケーションをGitHubAPIを利用したサンプルアプリとして構築します。

2017年4月の開発開始時では、まだまだiOSアプリでReduxを採用している事例は多くはなく、WebアプリのReduxではたくさん見かける開発ノウハウや情報もiOSアプリ開発では限られていました。1人で始まった開発ですがメンバー増加時のキャッチアップや学習コストを勘案し、WebアプリのReduxの情報も参考になるよう、できるだけRedux本来の思想や構成を踏襲するように実装することを心がけました。



● 図 15.1　GitHubAPI を利用した Redux のサンプルアプリ

紙面で紹介しきれないコードもたくさんあるので、ぜひサンプルコードのサイトからダウンロードしてお手元でアプリを起動し、それぞれの振る舞いがどのようなコードによって表現されているのか眺めてみてください。なお、本章に記載しているコードは、より解説内容に焦点を当てるために、一部サンプルコードと異なる場合があります。

Reduxのサンプルアプリは、次の機能を実装しています。

- パブリックなリポジトリ一覧の表示
- ログインユーザーに紐づくリポジトリ一覧の表示
- リポジトリ詳細の表示
- リポジトリのお気に入り管理（アプリ内で永続化して状態保持）
- ログイン、ログアウト
- アカウント情報の表示

画面上で多様な状態や振る舞いを取れるほうがReduxの強みをもっと活かせるため、リポジトリ一覧表示画面では次の非機能要件も実現します。

- 初回通信、プルリフレッシュリロード通信、暗黙的なリロード通信の使い分け
- 通信中、通信成功、通信失敗、サーバエラーの状態管理

リポジトリ一覧画面はIGListKit[注1)]によるUICollectionViewを利用し、アカウント情報画面はUIStackViewを利用してViewを構成しています。両者の実装の違いも見比べてみてください。

15.1 RxSwiftの活用

RxSwiftはFRP（Functional Reactive Programming）を表現できるライブラリです。サンプルアプリでは、データの変化とViewの変化の連動、非同期処理の記述表現の向上、非同期テストでの活用を背景に導入します。具体的には次の観点です。

- ReSwiftの購読機能の代替（BehaviorRelayで表現）
- ReSwiftのState変化のフィルタリング（filterやmapオペレータで表現）
- ReSwiftのState変化の検知（distinctUntilChangedオペレータで表現）
- ReSwiftのStateとViewの接続（Binderで表現）
- 非同期通信処理（Singleで表現）
- 非同期テストでの活用（RxBlockingを利用）

注1） https://github.com/Instagram/IGListKit

RxSwiftを中心にアプリを開発する場合では、状態をストリームで表現しビジネスロジックをオペレータを利用してFRPで記述するケースも多いですが、本章ではビジネスロジックにRxSwiftのオペレータの活用は極力しません。このようなビジネスロジック部分はReduxのState、Reducer、Action、ActionCreator、Middlewareを活用して表現します。

RxSwiftで一番難しい記述である、ビジネスロジックをFRPで抽象化して表現する部分をReduxが担います。そのため、RxSwiftで利用する機能や考え方は限定的となり、RxSwiftの上級者でなくても導入しやすい構成となっています。またReduxレイヤーのビジネスロジックおよび、ビジネスロジックに対するテストの記述は、RxSwiftを意識することなくPure Swiftのみで記述できるように配慮をしています。

● 図 15.2　GitHub のパブリックリポジトリの一覧を API から取得し表示する画面

本節では、GitHubのパブリックリポジトリの一覧をAPIから取得し表示する画面（図15.2）のサンプルコードを用いて解説します。この機能に関連する実装は次のとおりです。

- **View Controller**: `PublicRepositoriesViewController`
- **State**: `PublicRepositoriesState`
- **Reducer**: `PublicRepositoriesState.reducer()`
- **AsyncCreator**: `PublicRepositoriesState.requestAsyncCreator()`

15.1.1 ReSwift の購読機能の代替

ViewでRxSwiftのオペレータとバインダーを活用したいため、ReSwiftの購読機能をRxSwiftに置き換えます。次のような狙いがあります。

- RxSwiftのmapオペレータを活用し、Stateの局所や値に注目した記述ができる
- RxSwiftのdistinctUntilChangedオペレータを活用し、関心のある値が変化したことを検知できる
- RxSwiftのfilterオペレータを活用し、関心のある値が任意の条件に合致したことを検知できる
- RxCocoaのBinderを活用し、容易に変化する値をViewへ接続できる

ReSwiftの購読機能を代替するため、ReSwiftのStoreインスタンスをラップするクラス（リスト15.1）を設けます。Storeはアプリ内で1つだけインスタンスが作成され各画面のView Controllerが参照し、ReSwiftのみの実装では各View ControllerでStoreの購読を行い変化を検知していました。本サンプルでは各View ControllerはRxSwift.Observableを介して検知できるように変更します。

RxWrappedStoreクラス自身が、Storeの購読を行いStateの変化を検知します。検知された値をRxCocoa.BehaviorRelayに格納し、ComputedPropertyでRxSwift.Observableに変換して、Stateの変化を検知できるRxSwift.Observableを提供します。

Storeインスタンスは内部に隠蔽して、各View Controllerからは直接扱えないようにします。RxWrappedStoreクラスはReSwift.DispatchingStoreType protocolに準拠したので、外部からdispatchメソッドの呼び出しも可能です。したがって、RxWrappedStoreクラスは、Stateの変化をView Controllerから購読する機構と、Actionをディスパッチする機構を有したので、ReSwiftのStoreの代替として機能します。

● リスト 15.1　ReSwift の購読機能の代替 - RxWrappedStore

```
// ReSwift
// public protocol DispatchingStoreType {
//    func dispatch(_ action: Action)
// }

class RxWrappedStore: ReSwift.StoreSubscriber, ReSwift.DispatchingStoreType {
    var state: AppStateType { return self.store.state }
    lazy var stateObservable: RxSwift.Observable<AppStateType> = {
        return self.stateBehaviorRelay
            .observeOn(MainScheduler.instance)
            .share(replay: 1)
    }()
    private lazy var stateBehaviorRelay: RxCocoa.BehaviorRelay<AppStateType> = {
        return .init(value: self.state)
    }()
    private let store: ReSwift.Store<AppStateType>

    init(store: ReSwift.Store<AppStateType>) {
        self.store = store
        self.store.subscribe(self)
    }
    deinit {
        store.unsubscribe(self)
    }
    func newState(state: AppStateType) {
        logger.verbose(dumpState(state))
        stateBehaviorRelay.accept(state)
    }
    func dispatch(_ action: ReSwift.Action) {
        if Thread.isMainThread {
            store.dispatch(action)
        } else {
            DispatchQueue.main.async { [weak self] in
                self?.store.dispatch(action)
            }
        }
    }
}
```