[iOS] UIKit 100% だったプロジェクトの SwiftUI 化の終わりが近い話（参画時の面談から現在、そして）

どうも、@tobi462 です。

時は遡って、2024年11月にある iOS プロジェクトに参画しました。

そのプロジェクトは Objective-C コードこそ無かったものの、UIKit・RxSwift 100%（つまり SwiftUI や Concurrency のコードは 0）で Coordinator パターンが採用された、今から見ると1世代（あるいは見方によっては2世代）前のコードベースでした。

記事タイトルでは SwiftUI 化としていますが、もともとは普通に機能開発のために参画したプロジェクトで、機能が十分に拡充しサービスの事業ステージが変わった 2025年9月あたりから、 空いた時間で SwiftUI 化を進めていった結果気づけば大部分の画面の SwiftUI 化が完了していました。

プロジェクトを振り返ると、

- 普通にやるべきことを
- 普通にやって
- 普通にうまくいった

私の20年近いエンジニア経験の中で最初のプロジェクトだと感じています。

ただし、

- 開発は CEO を含めて4名の小規模チーム
- 私は1年以上 MTG なし（チームメンバーの顔や名前も知らない）
- Slack 上のコミュニケーションも驚くほど少ない
- ドキュメントもほぼ 0（API スキーマのみ）

という **「ここまで削っても大丈夫なのか」** と個人的な驚きも経験したプロジェクトでした。

一方で振り返ると（すべてのプロジェクトがそうであるように）改善すべき点もあったように感じますし、私のエンジニアリングの悪手もあったように感じます。

この記事では**プロジェクト参画時の面接から現在に至るまで**を振り返って、1エンジニアの経験則として残したいと思います。（細かいコードの話は別記事にしたいと考えています）

思いの外長くなってしまったので、**技術書の感覚**で読んでいただくのが良いかもしれません。

iOS プロジェクト構成

最初にプロジェクト構成が分かったほうが読みやすいと思うので、参画時の iOS プロジェクトの構成について書いておきたいと思います。

- サービス
  - EC 系（だけど少し特殊）
- UIKit 100%
- RxSwift 100%（≠ フルリアクティブ）
- Coordinator パターン（メモリ解放なし）
- API
  - APIKit
  - JSON:API
- ツール類（SwiftFormat や SwiftLint など）
  - なし
- ドキュメント
  - README.md と API スキームのみ

おそらく **「1世代前（RxSwift 時代）」** という表現がピッタリなプロジェクトかと思います。

RxSwift は入出力全体をつなぐフルリアクティブ （と私は定義しているのですが） ではないものの、コードベース全体にわたって RxSwift が使用されている状況で、Singleton に定義されたグローバルな値も各画面で購読されていました。

Coordinator パターンはメモリ解放が考慮されておらず、各画面から脱出しても ViewModel などがメモリに残りつづけ、さらに前述したグローバルな値の購読も解除していないため不可解なバグが発生しやすい状況になっていました。（辞書式の管理だったので同じ画面を（同じ導線で）表示した際は上書

きされ、永遠とリークしつづけることはありませんでしたが）

私にとって初めてだったのは JSON:API で、これは乱暴に言ってしまえば GraphQL のように必要な関連リソースを Client から指定して取得する API 仕様で、画面に応じて必要以上のリソースを API に要求しないのに役立っていたと思います。

SwiftFormat や SwiftLint などの周辺ツールは何も導入されていませんでしたが、プロジェクトに関わった人数が少なかったこともあってかコードスタイルにそれほどバラツキが無かったのは幸いだったかもしれません。

ドキュメントはメンテが追いついていない README.md と API スキーマのみで、つまるところ iOS プロジェクトにまつわるドキュメントは 0 でした。（開発プロセスなどの類も一切ありませんでした）

他に特筆すべき点としては、**現在の事業のサービス形態に落ち着くまで3回転**くらいしており、現在では使用されていない・あるいは限定的にしか使用されていないコードが多数残っていました。（最近 Backend 側の PR では「現在では何1つとして正しい内容が無いのでドキュメントを丸ごと削除」という PR が出されていました）

技術的負債の多いレガシーコード？

このように書くと「技術的負債を膨大に抱えた（1から書き直したくなる）レガシーコード」という印象を与えますが、私が今まで参画した iOS プロジェクトの中では（表現的に良くありませんが率直に書くと）**もっとも「マシ」だった**と最初に感じました。

その理由は **「事業のために書かれ続けたコードベース」だった** からだと思います。

それは

- RxSwift の機能は限定的にしか使っていない（Operator はおろか Single さえ使われていない）
- オーバーエンジニアリングされたコードは少なめ
- 他のプロジェクトよりはコメントが多い （といっても比較すればの話）
- 動作確認用のコメントアウトされたコードがある（現実主義 > 原理主義）

といった点から感じられたもので、おそらくこの例だけでは十分伝わらないと思うのですが、とにか

く**手段と目的の分離がハッキリしている**と私は感じました。

私が「問題のあるコードベース」だと感じるのは、

- 事業ではなく技術のために書かれたコード（オーバーエンジニアリング）
- 問題を放置しすぎて治安が崩壊したコード（技術的負債の爆発）

の大きく2パターンで、今回のコードベースは全体としてみれば中間地点に収まっている、私の中では「失敗していない」（＝成功している）プロジェクトのコードベースだと感じられました。

参画までの流れ

ここからは時を遡って参画前のタイミングから見ていきたいと思います。

参画のキッカケ

本記事で取り上げる iOS プロジェクトは稼働リソースに余裕があった時期にいわゆる「お手伝い」として参画した社内の別プロジェクトで、もともとは X社（仮名です）の新規サービスに参画したのが最初でした。

ことのキッカケは私が信頼を置いている iOS エンジニアさん（この記事は読まれているでしょうか…？）が業務委託を Twitter で募集しているのを見かけてその内容に興味をもったことでした。

そして DM で連絡して採用担当者さんと繋いでもらい面談の運びとなりました。

この時にその案件にピンと来た理由について思い返してみると、

- 新規サービスというチャレンジングな案件: 2割
- 信頼を置いている iOS エンジニアが所属している安心感: 3割
- 会社のホームページから感じられる（技術力ではなく）エンジニアリング力: **5割**

といった割合だったように感じられます。

ホームページにはその会社さんのサービス開発に対する哲学が書かれており、その内容が私のエンジニアリング感と一致していたのが最大の決め手だったように思います。

面談（疑似プロジェクト）

相手方の参加者は以下の3名でした。

- CEO
- CTO
- 社内の iOS エンジニア（Android もできる）

採用担当者さんから事前にこの構成を聞いて、「これは面接みたいになりそうだな」と思ったのをよく覚えています。（そして実際そうなりました）

面談は以下のように流れました。

1. 挨拶
2. 自己紹介
3. 疑似プロジェクトを想定した面接
4. 契約まわり

私の自己紹介ではスライドを使用したのですが、これは **以前 Twitter にも投稿した（気がしなくもない）「トビについて」** をベースにしており、面談時はこれに直近のプロジェクトの内容・実績を簡単に追記したものを使いました。

面談時にスライドで自己紹介するというのは直前のプロジェクトから始めた試みですが、個人的には短時間でどういうエンジニアなのか雰囲気を伝えられ、かつ **（経歴書を読み上げるような）硬い空気にならない** ので気に入っており、今後も面談のときはこれを使用していきたいと考えています。

面接は架空のスタートアップ的なプロジェクトを想定したものでしたが、**今回業務委託として募集されていたプロジェクトの性質そのまま**で、つまるところ今回の仕事でうまく振る舞えるかを問われたものでした。

これは感覚値になりますが、

1. デザインや仕様書がない状態でも開発を進められるか（自発性）

2. 状況に応じて切るカードを見極められるか（判断力）

の大きく2点を面接中で繰り返し問われていたように記憶しています。（個人的に 1. について何度も形を変えて確認されたのは意外でした）

わりと長い面接だったので内容をあまり覚えていませんが、おそらく一番緊張が走った局面は以下だったと思います。

CTO「このプロジェクトでテストは書きますか？CI/CDはいつ構成しますか？」
私「テストは必要になるまで書きません。CI/CDもやるかは状況次第です」

白状すると「攻めすぎたかな」と思いましたが、エンジニアとして自分が考えていることを素直に伝えたほうが信頼につながりますし、実際のところ面接で正直に話したことが参画後の信頼関係にもつながったように私は感じています。

契約条件（10日間の働きで継続判断）

先方から掲示された契約条件は

- まずは1か月間
- 参画して10日間ほどの働きで継続判断する

という内容でした。

新規サービスとは言え、まったく新しい環境でわずか10日間でしっかりとした成果を出すのはエンジニアとして難しく、他の会社に比べるとかなり厳しい条件だとそのときに感じました。（Google の研究では「既存メンバーと同じような生産性を出すまでには平均して9か月掛かる」と出ていたと記憶しています）

しかしこの条件を聞いたことで、私は本当の意味で先方のエンジニアリング力を信用したと思います。

私の持論として **「実際に一緒に働いてみない限りエンジニアの力量は分からず、面接は自社に適合しないエンジニアを採用してしまう確率を下げるフィルターでしかない」** というものがあり、そういう意味で本当にソフトウェア開発会社の経営について理解されてる会社さんだと思ったからです。

面談を終えて

面談を終えて、私の中では OK が出たら参画することに心を決めていました。

圧迫面接というほどではないものの、私の面接経験の中では過去一厳しく突っ込まれた面談だったと思いますが、

- 面白かった
- 楽しかった

というのが正直な感想としてありました。

疑似プロジェクトを想定した面接は候補者を判断するうえで有効な手法だと思いましたし、 CEO や CTO がどのように候補者を見抜いているのかという点も学びになり、技術的なディスカッションも非常に楽しかったと記憶しています。（私は面談が好きなのかもしれません）

そのあと先方から OK の返事があり、無事に X社の新規サービスに iOS エンジニアとして参画することになりました。

新規サービス開発（最小構成チーム）

この記事の本題ではありませんが、やや通ずるものもあるので新規サービス開発についても軽く触れておきたいと思います。（無事に10日目に継続をお願いされ、その後は3か月更新となりました）

チームは以下の4名構成でした。

- PM・PdM（CEO）
- デザイナ
- API・Backend
- iOS（私）

これは今回の記事で取り上げるチーム構成と似ているのですが偶然ではなく、X社ではサービスを上手に開発するための**メソッドとして「最小構成のチーム」を掲げている**ことに寄ります。（このあたりもエンジニアリング力が高いと面談前に感じた要因です）

CEO は面談の場だけだと思っていたので、実際に PJ の指揮を取る立場として参画されたのは驚きました。（「だから CEO からの尖った質問も多かったのか」と今では納得しますが）

ちなみにこのプロジェクトを含めて X社では3つのプロジェクト（うち1つは本記事では割愛したいと思います）に関わることになるのですが、開発マシンは私物の **MacBook Air M3 / 24GB** をずっと使っていました。（やや不足を感じなくもないものの許容範囲だったと思います）

この新規サービスは社内テストを2〜3回まわした結果、残念ながら CEO 判断によりクローズされました。

UI/UX も一般的なアプリよりも（ちゃんと考えた上で）攻めており、SwiftUI で実装するのも一般的なアプリよりはテクニカルで面白かったので、個人的にはそういった点でも少し淋しさを感じたのを覚えています。（プログラミングは手段と割り切っていますが感情は別です）

ところで、この開発期間中には API・Backend 担当者とペアプロする機会がありました。

「API まわりは詰まると時間が掛かるのでペアプロしませんか？」という API・Backend 担当者からの提案がキッカケだったのですが思いの外よい体験で、この **API・Client でペアプロする手法は今後も必要な局面で使っていきたい**と学びになりました。

子会社プロジェクトへの参画（と最初の失敗）

さて、ここからは本記事の主題にフォーカスして行きたいと思います。

前述したとおり開発リソースに余裕があったため、一時的なお手伝いとして参画することになったプロジェクトでした。（それが1年半もの長い付き合いになりました）

事前に GitHub リポジトリへのアクセス権をもらったので軽くコードリーディングを進めておき、それから参画する子会社の CEO（iOS エンジニア）との顔合わせ MTG がありました。

前述してきた（親会社の）CEO から紹介されるような形の MTG だったのですが、そのときに（親会社の）CEO から振られた **「トビさんから見てこのプロジェクトはこうしたほうが良さそうとかあったりしますか？」** という質問に対して、今でもたまに思いだす良くない受け答えをとっさにしてしまいました。

- ほぼ SwiftUI で作れるので生産性のためにも SwiftUI 化していったほうがいい
- ネーミングがちょっと気になるものが多い

これは本心から出た回答ですが、初対面の CEO に対してそのまま伝えてしまったのは **0点まではいかないにしても赤点**だったと思います。

私はエンジニアとして仕事するうえで **「信頼関係の構築」** が最重要だと考えており、比喩的な表現ですが「信頼関係が構築できてしまえば仕事の8〜9割は終わったようなもの」だと例えたりもしています。

その信頼関係を構築するうえで一番恐れているのは **「技術に傾倒して事業目線で見れないエンジニアだ」という印象を与えてしまうこと**で、この回答はそれを完璧にこなすものでした。（SwiftUI が得意・好きだからそれで統一したいエンジニアなのではないか…？といった）

しかしながらその後のディスカッションによるものか、あるいは事前に親会社の CEO からの前評判があったのかわかりませんが「SwiftUI を導入していく」という方向で話がまとまりました。

冒頭で記載したように UIKit 100% のプロジェクトであり、私も一時的なお手伝いとしての参画ということもあって、 **「SwiftUI で作っても今のチーム体制ではメンテできなくなってしまうのではないか」** という懸念も CEO から出ましたが、最終的には「SwiftUI が詳しい人がいるうちに導入を進めよう」という判断に落ち着きました。

あわせて、この時のアプリは iOS 15+ だったのですが、SwiftUI で生産性を出しやすい iOS 16+ にターゲットを変更することになりました。（当時としては iOS 15 を切っていないアプリは少ない方だったと思います）

なお現在も iOS 16+ で開発を進めています。

最初の機能実装・コードレビュー（SwiftUI 解説）

そんなわけで最初の実装となりました。

最初にしては結構ボリューミーで新規画面に加えて既存画面の改修も含むものでしたが、UIKit / RxSwift とのブリッジが少し面倒なくらいで、新規画面は SwiftUI で実装するうえで難しい仕様・デザインでも無かったので、技術的にはそれほど苦労しませんでした。

ただ、仕様や既存コードの意図は分かりづらいものもあり、このあたりはドキュメントが存在しない点も含めてレガシーコード・プロジェクトを感じさせられました。

さて、実装が終わって PR を出したのですが、そこで対面レビューをお願いされました。

前述したように CEO は iOS エンジニアで直近のコードはすべて1人で書かれていたのですが、SwiftUI についてはまったく知らない状態だったので、コードの内容も含めて SwiftUI について教えてほしいという趣旨でした。

説明しながら、

- SwiftUI って単純に見えて難しい部分もある
- UIKit 時代に比べて異質な書き方も多い

などと思ったのを覚えています。

私はもう SwiftUI に慣れてしまったので当たり前に感じてしまいますが、最初に SwiftUI の考え方や書き方を覚えるのに苦労したことを思い出させられました。

今ではそれこそ AI に聞けば何でも出力・教えてくれますが、それでも（SwiftUI に限らずですが）**プロジェクトに SwiftUI を導入するタイミングでは精通したエンジニアがいたほうがスムーズ**で、参考にできる既存コードが増えるに従ってチームメンバーの生産性も上がってくるのだろうと考えています。

これは前述した参画時の顔合わせを含めて2回目の MTG でしたが、これが私にとってこのプロジェクト最後の MTG になりました。

開発体制（小規模チームの強さ）

この施策を皮切りに次々と機能を開発していくことになりました。

事前に少し触れたように開発チームは4人体制で以下のような感じでした。（Android を除く）

- CEO 兼 PM・PdM（iOS エンジニアで私の PR をすべてレビュー）
- デザイナ

- Backend・API・Web（正確にはレビューアーなど含めて3人だがメインは1人）
- iOS（私）

つまり私が参画する前は、**たったの3人という超小規模チーム（CEO が iOS を担当）** で回していたことになります。

私はこのソフトウェア開発プロジェクトが相当うまくいったと評価していますが、**この小規模チームの体制が最大要因だった**と感じています。

”人を増やしても早くならない” というのは有名で、『人が増えても早くならない』 という直球な書籍（良書なのでおすすめです）も出版されているくらいですが、サービスの規模が大きくなってくると人を増やしたくなるのが人情というもので、それを自制してここまで小規模チームを維持できていたことには大変驚かされました。

ちなみに日々のコミュニケーション量も非常に少なかったのですが、参画期間が長くなって**信頼関係が深まるにつれてコミュニケーション量がさらに減った**のは印象的で、Issue・Figma・ API スキーマはどんどん簡素化してコミュニケーションコストの削減につながり、さらに生産性が上がったように感じています。（API レスポンスの例も途中から無くなりました）

これは **「ここまで伝えればあとはいい感じにやってくれる」 という線をお互いが分かってきた**ことに起因すると私は分析しています。

たまに Slack におけるコミュニケーションで「リアクションは返事じゃない」という議論を耳にしますが、このチームにおいては立派な返事として機能しており、議論すべき内容がなければ私も含めてみんなリアクションだけで完結させています。

おそらく **「相手が見たらあとは任せられる」 という強い信頼感**から来ているのだと思います。

開発プロセス（完璧でない状態を受け入れる）

開発プロセスはあって無いようなもので、

1. Issue を作ってそこでディスカッション
2. デザイナが Figma を作る

3. Backend が API を作る（必要なら）
4. iOS が実装する
5. PR を投げる

といった基本的な流れはあるものの、実質的にルールとして制定されたものは1つも無かったように感じます。

いわゆるスクラムに代表されるようなイテレーションも存在せず、

1. いい感じに開発して
2. いいタイミングで QA して
3. いいタイミングでリリース

という、スタートアップでさえなかなか無いのではないかと思うくらいシンプルなもので、スケジュールとしては「○月くらいにリリースしたい」みたいな大雑把なものだけでした。（このあたりは CEO がエンジニアというのが大きく影響しているのかもしれません）

これは私個人の考えですが、現代の高速化したソフトウェア開発においては**見積もりをするほうがコストになる**と直感しており、開発スピード（ベロシティ）に問題がなければ見積もりをしないに越したことはないのではないか、と近年では考えています。

開発プロセスとは直接関係ありませんが、**チーム全体に「完璧を目指すより、まずは完了を目指す（Done is better than Perfect）」の精神が暗黙的に根付いて**いて、 そこからさらに進んで**完璧でない状態を（それが自然だと）受け入れている**ような空気さえあったように感じられます。

これは様々な要因が組み合わさった結果だと思うのですが、この暗黙の精神・哲学はこのプロジェクトを成功させる上で大きな要因になったように感じます。（これは「Done is better than Perfect」などのコピーを掲げるだけでは到達できない領域だったように感じ、言ってみればチームビルディング（チーム醸成）の結果の産物で、再戦させるのは難しいようにも感じます）

SwiftUI 導入方針・設計

さて、おそらくエンジニア的には一番興味を引くであろう導入方針や設計についてですが、特筆すべき点はあまり多くないと考えています。

人数が多いチームでは認識や足並みを揃えるためにガイドラインなどを作ることもありますが（実際、X社の前のプロジェクトでは作っていました）、今回は私1人がすべてのコードを書くチーム体制だったのでそういったものは必要なく、これまでのプロジェクト経験からうまく機能したコードベース・設計を採用しただけとなっています。

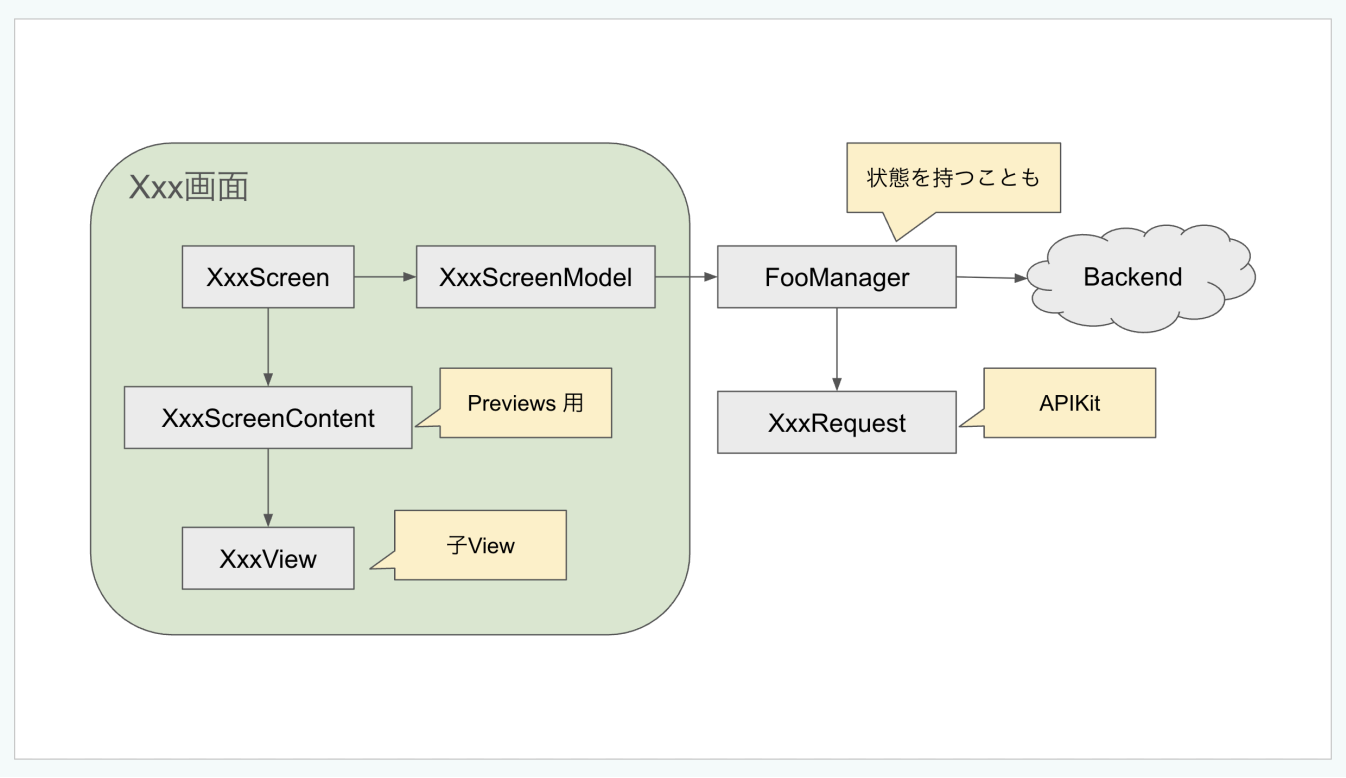
設計についてまとめると以下のような感じで、

- SwiftUI 系の補助ライブラリなし（Swift Algorithms のみ）
- View / ViewModel が基本構成（ViewModel は任意）
- API の窓口となる Manager（「お気に入り」などものによってはアプリ全体の状態も持つ）を用意
- Previews は基本書く
- EnvironmentObject は基本使わない（Singleton を Observed する）
- Environment も限定的に（現状でも3つくらい）
- 共通部品は使用頻度の高いもののみ（共通化を急がない）
- 画面遷移は UIKit ベース（既存コードにあわせて仕方なく）
- コメントをたくさん書く

これまで私が関わったプロジェクトよりもさらにシンプル化が進んだと感じています。

1画面当たりの基本構成

一般的な1画面の構成は以下のようになっています。



基本構成はこのイメージなのですが何点か補足すると、

- View に View っぽいものを書く（状態をもっても OK）
- ViewModel（図中は ScreenModel）に処理っぽいものを書く
- View / ViewModel はあわせて1つと見る（明確な分離はしない）
- API を叩かない（= Manager を使わない）なら ViewModel は任意
- Previews は仕様説明を意識する（あとから分かりやすいように）
- XxxManager は API の窓口としての役割がメイン

といったアーキテクチャ観点で言うとかなり緩いものになっています。

これは **「厳密な責務分割や統一された設計を目指すよりも、シンプルさを維持したほうが保守性は高くなる」** という個人的な経験則から来ており、画面や実装者によって書き方が多少バラけても問題ないと私は考えています。

例えば古典的には ViewModel にすべての状態をもたせたほうが責務分割視点では分かりやすく感じますが、View からのみ意識すればよい状態（遷移用の状態など）を ViewModel に定義すると、あとからコードを読んだときに行ったり来たりして逆に読みづらくなると感じており、仕様を表現する最小限のコードに寄せたほうがシンプルで保守性が高いと考えています。

ちなみに SwiftUI では ViewModel は不要（Model だけで良い）という議論もありますし、その主張も別に間違っているわけでもないと思うのですが、

- 画面固有の処理や状態をすべて View に寄せると複雑になる
- 画面に閉じた状態を保持するなら ViewModel がもっともスコープが狭い

と私は考えており、最初は ViewModel から始めて複数の画面で状態を共有する必要が出た時に XxxManager（でなくても良いと思うのですが）に状態を持たせるというアプローチを取っています。

仕様説明を意識した Previews

過去には「実行してさっと確認できる画面は Previews を作らなくても良いのではないか？（コスト的にペイしないのではないか？）」と考えて Previews を省略した時期もあったのですが、（自分で書いたコードですら）あとからコードを読んで修正する際に Previews があるかどうかで理解速度がかなり違うと自覚し、現在ではちょっと手間を掛けても Previews を書く方針にしています。

Previews は変更時のデザイン確認という側面が大きいと思いますが、それ以上に**仕様についての Overview を素早く提供してくれる**点により価値を感じており、仕様や表示パターンについて読み手により多くの情報を提供することを個人的には意識しています。

なお、API を叩いてそれを表示するだけといったシンプルな画面もときにはありますが、そういった時は Previews 用の XxxScreenContent を切らず、ただ XxxScreen を Preview するだけに留めています。（Previews を作ることが目的になると本末転倒と考えています）

EnvironmentObject

EnvironmentObject は使用しない方針を採っています。（そして現在のところ1つも使用していません）

それは **Singleton で定義して Observed して済むならそれが一番シンプルである**という単純な理由からです。

EnvironmentObject の設定ミスによるクラッシュは個人開発をしていたときも度々起こっており、それが複数人によるチーム開発ではより顕著になってストレスが溜まるだろうと個人的には考えています。

たしかに一部の Previews などで Singleton だとやりづらくなるケースも無くはないのですが、そのために EnironmentObject にするのはオーバースペックでトレードオフが釣り合わないケースが多いと経験則上は感じています。

もちろん決済などの一連の画面フロー中のみ状態を持って Singleton のようにアクセスしたいケースもあり、そういった状況において EnvironmentObject はコードの可読性を上げる道具として機能することもあると思いますが、それでも最初は StateObject を引き回す形から始めるのが、アプリ全体のシンプルさと保守性の維持につながると考えています。

Environment

Environment も本当に有効なケースを除いて利用しない方針にしています。

このプロジェクトで使用しているものは、

- シートを閉じるためのヘルパー
- フォームやコンポーネントのスタイル適用

くらいのもので、

- View の深い階層から気軽にアクセスしたい
- 横断的に一気に値を適用したい

というケースのみに留めています。

EnvironmentObject もそうですが、**コードをあとから読んだときにすんなり構造が頭の中に入ってくる程度のシンプルな利用に留める**のが、保守性とのバランスが取れていると感じています。

API 窓口としての Manager

私は過去に「API リクエストも各画面の ViewModel に書けばいい」と考えていた時期があります。

この考え方は今でも間違っていないとは思いますし、とくに新規サービスの開発初期においてこの割り切りは有用だと考えています。

ただ、コードを読み書きする際に具体的な API エンドポイント（XxxRequest）がさっと頭に出てこないケースが多いことに気づき、API 窓口あるいは名前空間として Manager を必ず用意するようにしています。

例えばお気に入り一覧画面を作成する際に、

- GetFavoriteRequest
- PutFavoriteRequest
- DeleteFavoriteRequest
- ...

といった個別のリクエストを考えて思い浮かべるより、

- FavoriteManager（に定義された関数〇〇）

の1つを思い浮かべる方が少なくとも私の脳にとっては負荷が少ないと感じています。（リソースの種類が多くなればなお）

また、特定の API を叩いたときにアプリ共通の状態を更新したいこともあり、それを個別の画面で行うと漏れが発生しやすくなるので、そういった点からも Manager を切り出したほうがトータルではプラスになるケースのほうが多いと現在は考えています。

シンプルな最小限のアプローチから始める

長々と書いてきましたが、どれも**シンプルな最小限のアプローチから始めることに終止している**という見方もできると感じています。

共通化や抽象化あるいは厳格な設計などは**静的な点として見ると**美しく優れているように映りますが、改修やバグ改修などのコード変更に代表される保守フェーズを含めた線として見るとそれほど優れておらず、却って保守性が落ちるケースもかなり多いと感じています。

実際、**私が関わった過去のプロジェクトではそういったものが大きな負債となって**おり、本当に必要なケースを除いてシンプルな実装を維持するのが保守性の良いソフトウェアにつながると考えています。

ちなみに Swift 言語の機能としても、

- Actor / AsyncStream
- マクロ
- Protocol
- Property Wrapper
- if / switch 式

といったものは（おそらく）1つも利用しておらず、そういった点でもシンプルになっています。

Swift は（分かりやすいので比較に出すと Go 言語と違って）強力な言語で色々できるため、ソフトウェアを開発する際は自分たちに必要なものを見極めて、必要以上に複雑にならないようにより注意が必要だと感じています。

スティーブ・ジョブズの電卓

ソフトウェア開発において「スティーブ・ジョブズの電卓」と呼ばれるテクニックがあり、これは色やデザインといったものを確認者（デザイナや PdM など）がその場で調整できるようにして、それを使って細かい調整をしてプロダクトの完成度を上げるというものです。（ちょっと大雑把な説明で恐縮ですが…）

SwiftUI で簡単な画面がすぐに作れるようになったこともあり、過去のプロジェクトを含めて私はデザイナや PdM に触ってもらう際に、

- アニメーションの種類・時間
- 挙動の種類
- 確認しづらいエラーケース

などを確認できるようなデバッグ画面を用意することが多かったのですが、このプロジェクトにおいてはとくに活用タイミングが多かったように感じます。

とくにアニメーションの種類・時間については言語化しづらい「サービスの雰囲気」という要素も大きく、このあたりの肌感覚はデザイナのほうがハッキリとした軸を持っていると感じており、たとえひと手間掛けたとしても色々試せるようにしたほうがプロダクト全体の価値が高まると考えています。

実際、デザイナさんから「今回も色々試せて助かりました！」といったコメントも頂いたりして距離感が縮まるのを感じ、チームビルディングや信頼関係の構築と言った点でも大きく機能したように感じます。

これに限った話でもありませんが、**「細かい調整というのは何度もリテイクしづらい」** ものでかといって自分が望むものを正確に伝えるのもそれなりにコスト（デザイナ側で期待するアニメーションデータを用意する、など）なので、そういった視点からチームのコミュニケーションコストを考えてみるのも有効だと感じています。

既存画面の SwiftUI 化

既存の UIKit 画面を SwiftUI 化する**純粋なリファクタリング**は参画してから1年弱ほど経ってから始まりました。

それまですべての開発リソースを施策に投じていたものの、EC・サービスとしての骨格が定まっていくにつれて施策も減る傾向が見えてきた為、その余剰リソースで既存画面の SwiftUI 化を進めることになりました。

それまでは、

- 新規画面は SwiftUI（稀にコピペ実装が早そうな場合は UIKit）
- 既存画面を改修する際はトレードオフ判断して UIKit のままか、SwiftUI で作り直し

というアプローチで進めていました。

トレードオフ判断についてはかなり経験則に依存していますが、「この画面は明らかに SwiftUI 化したほうが良い」と判断できるケースを除いて、UIKit のコード改修か部分的に SwiftUI を導入する形で済ませていました。

既存画面の SwiftUI 化についてあまり多く語ることはありませんが、

- 仕様やコードを十分に理解している画面から着手する
- リスクの高い画面や複雑な画面は後回しにする
- 既存のコードをしっかり調査しながら進める

- RxSwift などのレガシーコードの削除にこだわらない

といったことに気を掛けていたように感じます。

私の経験則上、既存のコード・仕様は長い年月で育てられているケースが多く、**一見すると意味の無さそうに感じるコード・仕様でさえ重要なことも少なくない**と感じています。

なので、SwiftUI 化する際は GitHub 上から blame して歴史を辿ったり `git log -p -S` なども活用して既存コードを十分に理解した上で、既存の処理・仕様をできるだけそのまま移植するようにしました。（仕様やデザインを整理・調整することはありましたが）

またせっかく SwiftUI 化するのだから「ついでに RxSwift も一掃したい」という気持ちも生まれるところですが、既存画面とのブリッジのために RxSwift の新規コードも遠慮なく入れました。（既存の ViewModel を一時的に残してブリッジさせる判断をした画面もあります）

結果として UIKit → SwiftUI 化の際に作り込んでしまったバグはわずかで、イベントログの欠損がいくつかある程度でした。（逆に言うと、**これだけ注意深く作業してもイベントログの欠損は起こってしまう**ものだと痛感しました）

ちなみに少し話はそれますが、**既存コードの調査をするうえで一番助かったのは PR に画面のスクショが残っている**ことで、これは当時の画面や仕様などを理解するうえでとても役に立ちました。（逆に言えばスクショが残っていない PR の調査は難航しました）

なので、たとえ開発が忙しいタイミングであってもひと手間掛けて PR に画面スクショを乗せておくことは、将来的な保守において非常に有利に働くとあらためて思いました。

Periphery

既存画面の SwiftUI 化を進めていくうちに、未使用コード・デッドコードがかなり残っていることに気付かされました。

前述したように移植する上では既存コードをかなり精査して行っていたため、「コードを読んで理解した挙句にデッドコードだった」というケースにも多く遭遇し、これは未使用コードの削除も並行で進めたほうが良いと判断して Periphery を導入しました。

とくに CI やビルドフェーズに組み込んだりはせず Xcode のターゲットとして単発実行できるようにしただけですが、未使用コードを探すうえではかなりの助けとなり、個人的に導入してよかったと感じています。（実際のところ導入するだけであればすごく簡単です）

ソフトウェア開発の理想的には使用されなくなったタイミングですべてキレイに削除できれば良いと思うのですが、私の中では **（Periphery などの）外部ツールの助けなしにそれをコスパよく行うのは困難** だと現在では考えていて、今後もデファクトツールとして採用し続けるだろうと思います。

SwiftFormat

コードスタイルを統一させる SwiftFormat ですが、**導入したのはつい最近**のことでまだルールもわずかしか追加していない状態です。

私は既存プロジェクトでも SwiftFormat をすぐに導入してしまうことが多いのですが、それは複数人の開発においてコードスタイルがブレて PR で指摘・ディスカッションになったりして時間を浪費することを防ぐためで、今回は私1人がほぼすべてのコードを書いていたので「将来的には導入することになるだろう」と思いつつずっと導入を見送っていました。

これは git blame が（1段階とはいえ）潰れてしまうのを嫌ったこともありますが、もっとも根本的なのは**事業的にそれよりも優先すべき開発がたくさんあった**という要因が大きいと思います。

ちなみに導入のトリガーを引いたのは Copilot による PR レビューで「ヘッダーコメントのファイル名と実際のファイル名が揃っていない」（あるあるだと思います）という指摘が増えてきたことに起因しており、いちいち指摘されるほうがコストと思い「SwiftFormat で自動修正するので対応不要」とコメントして後日導入する形になりました。

ちなみにビルドフェーズで行うと Xcode の undo スタックを壊して戻せなくこともあるので、単純に `swiftformat .` で手動フォーマットを行う形にしています。（実行漏れがあっても将来的にフォーマットされれば問題ないと割り切っています）

git commit にフックしたりあるいは CI で矯正する方法も考えられますが、このあたりも**まずはもっともシンプルなやり方から始める（課題が出たら増やしていく）** を心がけています。

ドキュメント・テストなし

ちなみにドキュメントは1つも書きませんでした。

これは参画時のプロジェクトがそうだったことにも起因していますが、私の中では**仕様書レスを受け入れる**という大きな心情の変化にもよります。

これについて語ると長くなるので別の機会にしたいと思いますが、コードにコメントをたくさん書いて読み進めれば仕様・ドメインの理解が深まるようにしておけば、仕様書を書かなくてもトレードオフ的にパスするという考え方をしています。

テストも（おそらく）1行も書いていません。

SwiftUI においては Single Source of Truth で実装するだけでかなりの品質が得られるため、むしろ変化が激しい現代ではテストを書いたほうがトータルでは遅くなるケースも少なくないと考えています。（これは参画時の面談でテストを書かない理由として補足した内容でもあります）

実際のところ過去に関わった iOS プロジェクトの多くはテストが負債になっており、かといってテストを消すのも不安でメンテコストだけが延々と掛かっているという場面にもよく遭遇しています。（このプロジェクトも含めて近年では消したテストコードの量のほうがはるかに多いです）

総じてドキュメントもテストも安易に作り始めるとそれを作ること自体が目的になってしまうので、プロジェクトの現状の課題感にあわせてトレードオフ判断することが大切だと感じています。（それこそドキュメントやテストに限った話でもありませんが）

技術的負債として立ちはだかったもの

このあたりで技術的負債として立ちはだかったと個人的に感じたものを紹介したいと思います。

プロトコルが多用された過剰設計

私の中でダントツでトップだと感じたのは Protocol が多用された過剰な設計でした。

本来は純粋な関数コールでさえ十分な実装に、Protocol やミュータブルな設計、継承を用いたオーバーライドなどが組み合わさっており、未使用コードやデッドコードを削除した現在でも全体像はよく分からないと感じます。

過去に参画した iOS プロジェクトにおいても同様のものが巨大な負債として登場しており、

- Protocol を廃止したいけれど事実上無理（なので延々と拡張する）
- 作り直したくても既存の処理を読みきれない

といった状況に陥っていることが多いと感じています。（実際このプロジェクトでも廃止できていません）

Protocol などを使った抽象化や共通化は**かなり慎重にカードを切る必要がある**とあらためて感じさせられました。

RxSwift

これは一定仕方なく思いますが、RxSwift のコードも負債として感じられました。

前述したとおりこのプロジェクトにおける RxSwift は控えめでしたが、それでも関数コールで済むようなところをストリームのチェーンとしてつなげている箇所もあり、既存コードの処理を読み解くうえでは負荷になったように感じます。

また Periphery でも未使用コードの検出が難しく、購読のみで書き込みが一切ない Rx 変数は気づいたタイミングで手動で削除する必要がありました。

Coordinator パターン

前述したようにこのプロジェクトの Coordinator パターンはメモリ解放が考慮されておらず、それが予期しない不具合につながることもあったのですが、それを差し引いても Coordinator パターンは負債として感じられました。

それは Coordinator のメモリ管理が画面階層にあわせたチェーンを形成しているためで、チェーンの一部だけ Coordinator をやめるようなことができず、画面の SwiftUI 化はかなり進んでいるのに Coordinator はいつまで経っても取り除けないという状況が続きました。

結局、Coordinator を剥がそうとして不具合やメモリリークにつながっても面白くないので、一連のすべての画面が SwiftUI 化されてから取り除く（無理に消すのを急がない）という意思決定をしました。

ところで過去の PR を見ていて Coordinator パターンが最初に導入された PR を見つけたのですが、CEO から「私は Coordiantor パターン使ったことないです！必要なんですか？」というコメントが書かれており、それに対するディスカッションもなく**なし崩し的に導入されてしまった**ように映りました。

たしかに議論しすぎて進まないのも良くないので難しいですが、**最低限の ADR（Architecture Decision Record）は残しておく**と将来的にトレードオフ判断がしやすくなると感じます。（ぶっちゃけ「流行ってたので入れた」みたいなのでもいいと思います）

コメントの少なさ

前述したように他プロジェクトに比べればコメントは多かったものの、それは比較すればの話であってトータルで見るとやはり少ないと感じています。

長年エンジニアとして仕事を続けてきて、私自身は **「コメントの量と丁寧さがもっとも保守性に影響する」** と最近では感じています。

そのため現在ではドキュメンテーションコメントはもちろん、ちょっとでも一瞬で読み解きにくいコードがあればコメントを書くようにしており、新しい参画メンバーでも「コードリーディングを進めれば仕様やドメインの理解が深まっていく」というのを理想にしています。

AI 活用について（ChatGPT / Copilot）

最終的に活用した AI は、

- ChatGPT
- Copilot（PR レビュー）

の2つだけでした。

iOS アプリ開発においては **「これどうやってやるの？」を調べる時間の割合がかなり多い** こともあり、ChatGPT で一発で回答・サンプルコードが得られるようになったのは、それだけでも私の生産性に大きな影響を与えたと思います。（Google 検索も StackOverflow も使わなくなりました）

Issue に書かれたビジネス用語なども一瞬で適切な回答が得られるので、そういった点においても ChatGPT だけでも完全にゲームチェンジャーになっていたと感じます。

コード生成はそれほど積極的に行っていませんでしたが、現代の Swift / SwiftUI のような環境ではコード自体の表現力が高いこともあって、わざわざ指示を出してコードを生成するより手で書きながら理解したほうがよいかな、という直感を持っていました。（一般的な処理や構造体などはコード生成のほうが早いと感じていましたが）

Copilot による PR レビューはプロジェクトの途中で導入されたものですが、エンジニアがコードレビューするよりも精度が高いのではないかと感じられ、正直なところ **Copilot の指摘を全部直せばあとは動作確認してマージで良いのではないか**と個人的には感じさせられました。（ただ、今のところ精度にバラツキはあって人間なら拾える、というのも残っている気がします）

一方、最近はかなり細かい点まで大量に指摘してくるのは難点で、このプロジェクトにおいてはオーバースペックな指摘が目立つようになってきたことで最近は「YAGNI」とコメントを返すケースが増えてきており、もう少し事業コンテキストを与えてあげる必要があるのかも、と感じたりもします。

また「YAGNI」コメントしてスルーしても別の PR（リリース用など）でも重複して指摘されるものストレスで、**元 PR で検出されなかった問題が新たに見つかる**こともあってすべてスルーするわけにもいかず、このあたりは今後の課題だと感じました。

ChatGPT を盲目的に信じたゆえの判断ミス

AI がすべての面においてプラスに働いたかというとそうではありません。

ある時、商品セルの impression を計測する施策があり、「現代だとどういう実装パターンが主流なのだろう？」と思い ChatGPT で調べてみましたところ、セルの可視領域まで考慮した賢い実装方法と完全なコードが得られました。

私はその時に、

「一発でこんな完璧なコードが出てくるなんて、**今はこれが一般的な実装パターンとして定着しているんだな。（経験則上だとスクロールパフォーマンスに影響しそうだけど…）**」

と思いました。

結果、実際にスクロールパフォーマンスに影響を与えるものだった上、私は最初のバイアスを引きずったままだったので実機で動作確認しても「まぁ許容範囲か？」となかば納得し、そのまま PR レビューまで出してしまいました。

CEO が動作確認した結果の「ちょっとカクつきます」という言葉でようやく目が覚め、頭を下げて作り直しました。

これは私にとって間違いなく**このプロジェクトにおける最大の判断ミス**で、そもそもセルの可視領域を考慮した impression はオーバースペックで、最初からそんな凝ったものを作る必要は無かったのです。

今思い返しても「なぜそんな愚かな判断をしたのか」と思いますが、一度バイアスが掛かると脳はそこから抜け出せないのだとあらためて自覚させられた一件でした。

やり直せるとしたら？

このプロジェクトを最初の参画タイミングからやり直せるとしたら、何を変えたいと思うでしょうか。

チームメンバーとの顔合わせ

まず、**チームメンバーとの顔合わせはやったほうが良かった**と私自身は思います。

たしかに一緒に働いているうちに自然とチームビルディングされ、お互いの顔や名前や趣味嗜好を一切知らなくても仕事には何の問題も無いことが分かりましたが、それでもコミュニケーションする上ではお互いの顔や雰囲気を知っていたほうが円滑だったように感じなくもありません。

住所フォームの共通部品

いわゆる住所フォームがいくつかの画面にあったので、

- 現在地から入力
- 逆引き
- 候補リスト
- バリデーション

といった典型的な機能を持たせた、（このプロダクトの中では数少ない）いわゆるステートフルでリッチな View を共通部品として実装していました。

これによって各画面で利用するのは楽になりましたし、実装ミスによる不具合も防げたと思うのですが「やや過剰な実装で複雑さをもたらしてしまったかな」と今でもこの判断が正しかったか迷います。

私のなかで答えが出ていないものの1つですが、おそらく次回に同じようなケースに遭遇したときは「コピペ実装したほうが良い」と判断する気がします。

普段、エンジニアとして正しい判断をできているか確認するために、自分の心の動きを見ることがあるのですが、

- 実装が楽しい・面白い
- いいコードが書けている
- これは記事にできそうな内容だ

といったような感情が出た時は「警告信号」として扱っており、あとから振り返るとこの実装をしたときは多少こういった心理状況が入っていたような気がします。

他にはあまりない？

他には…　と続けて書こうとしたのですが思いのほか出てこないですね。

大きな施策の場合に MTG で軽く口頭説明があっても良かったかもと思い返すこともありますが、それでも絶対にあったほうが良かったように記憶しているものはなく、やはり MTG は基本なしで良かった

のかもしれません。

技術的には細かい話ですが、Color の extension にプロジェクトの名前空間を設けるのを忘れたことを今でも後悔しています。

```
// Color.xxx ではなく
extension Color {
    static let xxx: Color = ...
}

// Color.project.xxx にしたかった...
extension Color {
    struct project {
        static let xxx: Color = ...
    }
}
```

幸い標準の色名と競合していなかったので困りませんでしたが（そしてそれ故発見が遅れましたが）、将来的に競合する可能性も考えると名前空間を切っておくべきだったと思います。

これを直すのはおそらくかなり面倒で、おそらくトレードオフ的に行われることは無いと予想しています。

おわりに

これまでの内容と重複しますが、このプロジェクトの成功要因は以下だったと思います。

- 小規模チーム
- 事業目線で動く
- 完璧でない状態を受け入れる
- クラフトマンシップ
- もっともシンプルなものから始める

こういったソフトウェア開発マインドというものは AI 時代になっても変わらないどころか、AI で色々

早く作れるようになったからこそより大切になってくる要素だと私は感じています。

ところで私の中でバイブル的な立ち位置となっている技術書として、

- Joel on Software
- 達人プログラマ
- ハッカーと画家
- DDD エヴァンス本

などがありますが、これらは**実際のソフトウェア開発から得られた（知識にとどまらない）経験が記されている**ことの価値が非常に大きいと感じています。

私もいつか実際のソフトウェア開発から得られた経験則を共有したいと思いつつ、所属企業にスポンサーされた技術記事でこういったものを扱うのは難しく、また正直に言えばこうしたエンジニア的な考え方を記事としてシェアすることに心理的抵抗もあり、いつまで経っても書けない状況が続いていました。

今回、それにチャレンジしてみた結果がこの記事となります。

この記事から何かしら得られるものがあればソフトウェア開発者として冥利に尽きます。

卒業

開発すべき施策も少なくなり、SwiftUI 化の終わりも見えてきたこともあって、**今月（4月）でこのプロジェクトを卒業**することになりました。

1年半の開発期間の中では **TVCM やメディア露出（TV で紹介）** など、サービスの事業ステージが変わったと肌で感じられるタイミングにも巡り合え、1エンジニアとして良い仕事ができたと満足しています。

来月（2026年5月） からは久しぶりに仕事が空きますので、もし業務委託のお仕事のご相談などありましたら @tobi462 までお気軽に DM いただけますと幸いです。（別途 X でも告知したいと思います）