チームで育てる Android アプリ設計

著者　釘宮 慎之介　横幕 圭真

編集　日高 正博

PEAKS

チームで育てる Android アプリ設計

著者　釘宮 慎之介　横幕 圭真

編集　日高 正博

PEAKS

はじめに

PEAKS から「Android アプリ設計パターン」が出版された時期は 2018 年 1 月 31 日です。当時の Android アプリ開発界隈では、そこかしこでアーキテクチャについての議論がなされていました。そして今日まで約三年の月日が流れ、現在ではアーキテクチャの議論は落ち着きをみせています。当時の議論は少なからず Android アプリ開発界隈のスキルの底上げに一役買ったはずです。

本業も副業も複数のプロジェクトに関わることが多い筆者の仕事柄、さまざまな現場を見て回る機会があります。そこには当時の議論の果てにたどり着いた理想のアーキテクチャがあるのかというと、そういった現場はめったにありません。これは現場のエンジニアが、アーキテクチャについて興味がないわけでも、知らないわけでもありません。

知っているけど、いざ自分のプロジェクトでどう構築すればよいのかわからない。導入しようとしても、中途半端になる。チームに浸透しない。

本書は、そういった課題の解決に向けて手助けします。

筆者は本書の第一部である第 1 章から第 4 章までを担当しています。特に新規開発にフォーカスをあてています。選定の基準などをリードするアーキテクチャ設計者が何を考えて構築するのか、アーキテクチャやアプリの開発基準をどうチームに浸透させていくのか、実例を交えつつ解説しました。

新規開発時のアーキテクチャ構築は、相当のスキルを要しますし、今後の事業を左右する重要なものです。構築したアーキテクチャによって、それ以降の「成長痛」の度合いが決まってきます。その一方で技術者にとっての新規開発は多くの知見を手に入れられる面白いタイミングでもあります。

本書が読者のチーム、事業、そして読者自身にとって最適なアーキテクチャ構築の助けになれば幸いです。

（著者 釘宮 愼之介／@kgmyshin）

Android というプラットフォームが登場してから 10 年以上が経ちスマートフォンが生活に溶け込んで、Android アプリが日々の生活に欠かせない存在になっている今、大小さまざまな組織で Android アプリの開発が行われています。

プラットフォームの進化とともに Android アプリ開発を取り巻く環境も劇的に進化しており、同時に開発組織もどんどん発展してきています。技術の進歩によって Android アプリでは、よりリッチで多様な体験を実現できるようになったことで、Android アプリを作る組織も少しずつ、あり方を変化させているように感じます。

本書は日々進歩し続ける技術とチームを一対と捉え、チームとして技術を磨いていくことと、進歩する技術をもとにチームを磨いていくことの双方を学べる本にしたいという思いをこめて「チームで育てる Android アプリ設計」と名付けました。

筆者は特に大規模な組織における Android アプリ開発の経験を踏まえて第二部の第５章から第８章の執筆を担当しています。国内でもまれにみる規模の開発組織でプロダクト開発と技術課題に向き合う組織について解説しました。Android の長い歴史から俯瞰すると、この大規模な組織での経験は執筆時点でまだ１年半ほどと短いものですが、それでもプロダクト・組織・技術それぞれが進歩し変化します。そのたびにチームとしてそれらの変化に向き合い続けてきました。

アーキテクチャは組織の大小を問わずチームでプロダクト開発を進めるための重要なツールのひとつです。アプリでのユーザー体験や機能を実現しようとしたとき、チームはアーキテクチャにしたがって実現方法を考えますが、プロダクト・組織・技術が進歩すれば当然、その見方や捉え方も少しずつ変わっていきます。本書では少しずつ変わっていく捉え方とアーキテクチャを整理・調節してチームとして付き合っていく上手な方法を学ぶことができます。

もしかすると筆者が体験したような大規模な組織での Android アプリ開発は他に例をみず、読者にとってはオーバースペックな解法であるかもしれませんし、本書に書いたことが正解であるとも限りません。ぜひ本書の内容をもとに、自分のチームで真似できそうなことや異なるやり方でよい結果が得られそうなことを議論し、読者自身にあったアーキテクチャの育て方を発見する手立てとしてください。

（著者 横幕 圭真／@KeithYokoma）

対象読者

Android アプリの開発に取り組むチームとアーキテクチャの両面を取り上げています。次のようなことに興味のある方を対象としています。

- アプリ開発をこれから始める方
- チーム開発をよりよくしたい方、技術的なチームビルディングに興味がある方
- アーキテクチャの選択、設計の維持に難しさを感じている方
- 新規開発／継続開発などチーム開発に関わる方

本書の手引き

本書『チームで育てる Android アプリ設計』は新規開発と大規模開発の 2 部構成です。第 1 章から順番に読み進めることを想定しています。次のまとまりや章単位で読むこともできますが多少の前後関係があります。

第 1 章〜第 4 章では新規開発を整理し、チームに合った設計を取捨選択する手法や事例を解説します。第 5 章〜第 8 章では大きな開発チームでの設計の役割と改善に焦点を当てます。生産性を維持しながらチームの抱える技術課題を継続的に改善する事例について議論します。

最後に第 9 章では新規開発と大規模開発の違いに触れます。新事業へのチャレンジとなる新規開発と運用の歴史を経ている大規模開発という 2 つのフェーズの差について理解を深め、開発に貢献できるエンジニアの振る舞いについて視点の違いからヒントを得ます。

想定環境

本書では表 1 の環境で動作確認をしています。

●表 1　想定環境

Android Studio	4.1.3
Kotlin	1.4.30

クラウドファンディングと PEAKS

本書は技術書クラウドファンディング・サービスである「PEAKS」のプロジェクトとして開始され、732 人もの支援者のサポートによって作られました。出資者特典である「アーリーアクセス」でいただいたご意見も反映されております。

PEAKS ではこんな本を作りたい！という方を募集しています。次の窓口からご連絡いただければ幸いです。

- https://peaks.cc/requests

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。

目次

第 1 章

新規開発とアーキテクチャとチーム

釘宮 慎之介／@kgmyshin

本章では新規開発とはどういう状態なのかを整理します。そして、新規開発の中で何も準備せずに開発を進めていくチームと、しっかりとアーキテクチャを用意をしたチームのふたつで生産性にどれだけの違いがでるかを説明していきます。

1.1 新規開発は○○がない

新規開発は何も決まっていない無秩序な状態から始まります。決まっている事柄があるとしたら、初期メンバーがアサインされているということでしょうか。あとは「なんとなくこういうものを作りたい」という要求もあるでしょうがフワッとしているはずです。そして、いつまでにリリースするという締め切りに関しては大抵、決まってしまっているのがほとんどでしょう。

そんな無秩序から始まる新規開発において、改めて何がないのか整理してみましょう。

1.1.1 コードがない

当たり前ですが新規開発の開始時には、まだコードはありません。途中から参加する既存プロジェクトでは、すでに存在しているコードとの統一性を考えたコーディングをするメンバーがほとんどです。間違いなく一定の経験があるエンジニアはコードの整合性を気にします。

しかし新規開発では、その指針となる既存コードがありません。開発のスタートを急ぐあまり「よーいどん」で一斉に書き始めると、統一性のないコードがプロジェクト開始直後に生まれてしまいます。

1.1.2 面識がない

新規開発には人々の面識がありません。もちろん今まで仕事をしてきた仲間と一緒に始める場合もあるでしょう。そのような恵まれた場面は決して多くありません。基本的には面識のないメンバーや面識はあるものの同じチームでの開発は未経験のメンバーと働くことになります。

一緒に仕事をやったことのない人々と仕事をする場合、相手がどのような思想の持ち主かは当然わかりません。どういうスキルレベル感で、どういうコードを好んで書くのか。少なくともエンジニアは誰もがプログラムの書き方についての好き嫌いを持ち合わせています。中には決して譲れないポリシーを持っているエンジニアもいます。この譲れないポリシーが衝突した場合、後々のプロ

ジェクト開発に悪影響がでるコード（負債）を残したり、本来不要な軋轢がチームで発生したりするのは必至です。

1.1.3 ルールがない

新規開発にはルールがありません。実務で必要となる仕様決定フロー、デザイン作成フロー、ブランチ戦略、命名規則など開発のための合意をとらなければなりません。

その中でも、アーキテクチャは新規開発を始める前に特に考えなければいけない重要なルールのひとつです。特にチーム開発におけるコード品質とチームの生産性を高める道具として機能します。

1.1.4 まとめると、新規開発には秩序がない

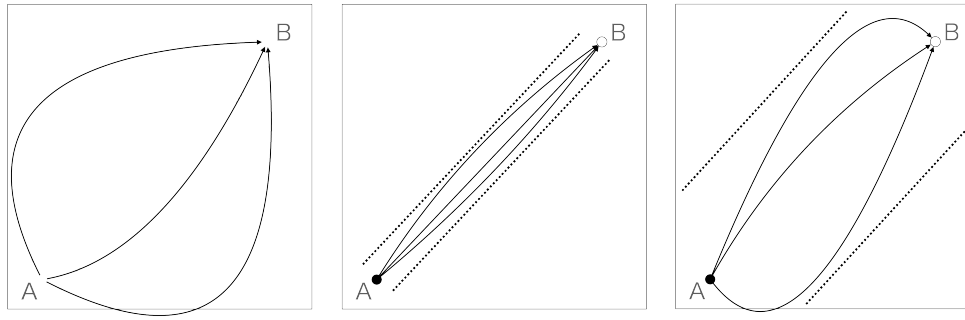
新規開発とは前述のように何もない状態から開発をしていくことになります。そして、一言にまとめると秩序というものが一切ありません。

開発を急いで準備をしない無秩序のままスタートしてしまうと、リリースにクリティカルな影響を与えるような問題が起きる可能性が増大してしまいます。本項ではこれ以上、取り決めるべき秩序の内容に触れませんが、アーキテクチャの他には次のような要素を準備しておくとよいソフトウェア開発が行えるでしょう。ソフトウェア開発のなかでも特にエンジニアが触れるコードに関わるものを列挙します。

- コードの統一性を確保する Lint 等、フォーマッタの設定
- コードレビューに備えた静的解析結果等をコメントする自動化の設定
- プロジェクト構成を管理する Gradle の整理（マルチモジュールの場合にバージョンを一元管理するかどうか、など）
- 継続的インテグレーションツールのセットアップや CI サービスの設定
- 継続的デプロイのためのツール、CD サービスの設定
- Git 開発フロー整備を目的としたメッセージテンプレート作成（Issue や Pull Request、コミットメッセージ）
- チームメンバーの協業を促進するブランチ戦略

1.2 アーキテクチャがないチームとあるチーム

大きくつまづかないためには、いろいろな準備が必要ですが、その中でも重要な要素がアーキテクチャの決定です。具体的なルールでは個別の議論に陥りがちとなるため抽象的な例えで解説します。次の図 1.1 をみてください。



●図 1.1　A から B への到達経路

A から B へのつなぎ方に制限がない場合、なにも用意しなければ平面上での A から B へのつなぎ方の選択肢は無限に存在します。人の数だけ方法がある無秩序な状態です。ここでガードレールのようなものを用意することができれば、ある程度の方向性、秩序を与えるられていることがみてとれます。しかし、そのガードレールの設置の仕方次第では、ほぼ無秩序な状態と変わらない状態にもなってしまいます。

アーキテクチャは、チーム開発においてこのガードレールのような役割をします。チームに必要な制限を加えるための幅が存在しています。実際の理解のために具体的なコードを交えながら、アーキテクチャがないチームとあるチームでどういった差が生まれるのか学んでいきましょう。

1.2.1　アーキテクチャがないチーム

アーキテクチャの適用について 2 つのケースを紹介します。まったくアーキテクチャを決めずに開発に走り出していくケースと、MVVM で行こうとだけ決めたケースの 2 つをみていきます。後者は一見アーキテクチャが**ある**チームに見えますが、実質ないに等しい状態といえます。

ここでは、想定読者を MVVM や MVP などがどういったものか知っており LiveData や ViewModel がどのようなものか知っている状態と設定します。もし MVVM や MVP がさっぱりわからないという方は PEAKS から出版されている Android アプリ設計パターン入門[注1)]を読んでみるとよいでしょう。

まったく決めていなかったケース

新規開発時にアーキテクチャを定めずにスタートしてしまうチームは、極端な例では次のような事態に陥ることがあります。

注1)　Android アプリ設計パターン入門: https://peaks.cc/books/architecture_patterns

- ある人は特定の機能を実装するために Redux を用いる
- ある人は特定の機能を実装するために MVVM を用いる
- ある人は特定の機能を実装するために MVP を用いる
- ある人は特定の機能を実装するために Activity や Fragment にすべて記述する

実装者によってアーキテクチャが根っこからバラバラになるという事態が発生します。こうなってしまうとなかなか取り返しがつきません。この状態から統一性を手に入れるには、かなりの苦労を要します。

既存のコードに統一性がある場合は、一定以上の経験を持つエンジニアはそれに従うと前述しました。従う動機は明確です。経験がアーキテクチャの重要性を理解しているからです。

しかし既存コードに統一性がみられない場合、また統一性を導き出せるほど整理されていない場合、人によっては自分の効率性を最大化します。第三者からは個人ごとの最適化の結果ではなく好き勝手に書いてしまっているチームだと感じるでしょう。

筆者の経験として、エンジニアは同一人物で変化がないのに、バラバラな既存の構造に合わせて機能ごとに実装方法を変えてしまうケースを観測したことがあります。器用にアーキテクチャを使い分けていますが実装しているエンジニアが別人になったわけではありません。非効率であると理解しながら、すでにある構造に従ってしまうのです。

こういった文化が蔓延したプロジェクトやチームでは、メンバーのマインドから変えなければなりません。チームでリファクタしていこうと決定したとしても、リファクタしている隣で、カオスな書き方のまま新規機能の実装が行われてしまい、改善が進まず挫折することはよくあることです。

一度、チームに根付いた文化を変えることはなかなか容易なことではありません。良いことであっても悪いことであってもです。

首尾よくマインドが切り替わったとしても先々の道のりは重労働です。コードの統一性を手に入れるためにはアーキテクチャを定めて、外れている既存コードを書き直す。すべて終わるまで繰り返す、これに尽きます。

文章で示すとシンプルですが**書き直す**ことにかなりの労力を要することになります。一度実装したところとはいえ、検証やリリースを含めると、これまでそれらの機能を実装した時間と同程度の時間がかかることが多いです。初めからアーキテクチャを定めて守ってさえいれば「書き直す」という大きな手間はなかったはずです。

MVVM とだけ決めたケース

前述の例ではまったく指針を決めず、それぞれが自由にアーキテクチャを選択してしまうという極端なケースを示しました。ここで例示するチームでは「アーキテクチャは MVVM で行こう」とだけ決めているとします。

こういったチームでは先と比べてどうでしょうか。結論から述べると、これだけでは先と同様ま

でいかなくとも近しい状況まで陥ってしまいます。

実際にコードをみてみましょう。異なる開発者がそれぞれ実装した ArticleListViewModel と ArticleViewModel というクラスはリスト 1.1、リスト 1.2 のように記述されています。

●リスト 1.1　記事一覧を読み込む ArticleListViewModel

```
class ArticleListViewModel(private val repository: ArticleRepository) : ViewModel() {
  fun fetchArticleList(fragment: ArticleListFragment) = viewModelScope.launch {
    val articleList = repository.findAll()
    fragment.submitList(articleList)
  }
  fun addArticle(fragment: ArticleListFragment, form: ArticleForm) = viewModelScope.launch {
    val article = repository.store(Converter.convert(form))
    fragment.moveToArticle(article)
  }
}
```

●リスト 1.2　記事を読み込む ArticleViewModel

```
class ArticleViewModel(
  private val id: ArticleId, private val repository: ArticleRepository) : ViewModel() {

  private val refresh = MutableLiveData(Event())
  val livedata = refresh.switchMap { r -> liveData { emit(repository.findById(id)) } }

  fun onSwipeRefresh() {
    refresh.value = Event()
  }
}
```

この 2 つのコードが同一プロジェクトにある場合、途中参加したエンジニアは何を思うでしょうか。

- 前者のコードはメソッド名が命令名になっているのに対して、後者のコードはメソッド名は on イベント名 となっている
- 前者のコードは View へのフィードバックには引数として Fragment を受け取り、Fragment のメソッドを呼ぶようにしている。後者は一切 View に関する情報はもたずにフィードバックが欲しい場合は LiveData を使用する思想で書いている

短いサンプルコードだけでも、この大きな違いが生まれています。

MVVM を構成するコンポーネントの中でも役割が人によってブレにくい ViewModel でさえ、このありさまです。エンジニアが途中参加してきた場合、何をもって書き分けをしているのか理解することは難しいでしょう。アーキテクチャの論拠がわからなくなり、コードリーディングの効率が大幅に落ち、コードを書き始めるまでに多大な時間がかかります。

さらに、このようなコードを放置しておくと次のようなコードが生まれます（リスト 1.3）。

第 2 章

アーキテクチャの選定と背景

釘宮 愼之介／@kgmyshin

第 1 章ではチーム開発におけるアーキテクチャの貢献度を説明しました。さらに適切な制限が威力を発揮させることにも触れました。本章では適切な制限をどのように設けるかを解説します。

2.1 背景が違えば選ぶアーキテクチャや使う技術が違う

前提として、アーキテクチャには普遍の正解がありません。チームメンバーやプロジェクト、背景によって選ぶべきアーキテクチャや使用する技術（アーキテクチャを含めたライブラリなどのさまざまな技術的要素）は変わってくるからです。では、どの背景でどういうアーキテクチャや技術を選定すればいいでしょうか。これについても明確な答えはなくケースバイケースといえばそれまでなのですが、ここは一歩踏み込んで考えてみることにします。

2.2 背景の要素

背景とひとことで表現してもチームが抱える課題はさまざまです。言葉に含まれる要素を洗い出してみましょう。筆者は次のような要素が主であると考えています。影響を与える度合いが大きな順で並べてみました。

- チームメンバー
- 締め切り
- 主流を追う
- 会社のルール
- Web API

ひとつひとつ説明していきます。

2.2.1 チームメンバー

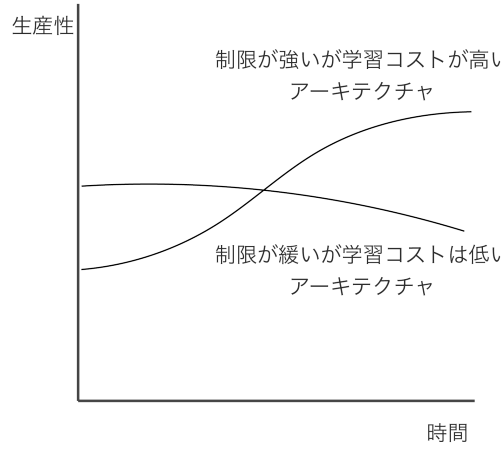
新規開発時に、どういう**チームメンバー**で開発を行うべきか。これが一番大きな変数です。それぞれのメンバーの習熟度という観点で説明してみましょう。

チームメンバーが、これまでどのアーキテクチャに慣れ親しみ、何を嫌っているか。モチベー

ションが湧く技術の傾向があるか。こういった個人的な性質やスキルの観点を踏まえてアーキテクチャを決めるべきです。

チームメンバー及びリード（の立ち位置にいるエンジニア自身）の習熟度を無視してしまうと学習コストが跳ね上がります。筆者は学習コストを無視したまま開発に着手してしまい、数ヶ月経っても一切進捗しなかったプロジェクトを見たことがあります。そういった事態に陥らないためにも、チームメンバーの習熟度に見合ったアーキテクチャ選定もしくは学習コストを織り込んだスケジュールを立ててみましょう。

前章でも触れていますが適切な制限を設定したアーキテクチャは生産性を大きく向上させます。ただし制限を多く設けるほどに複雑になり、学習コストは高くなってきます。そのため図 2.1 のようなグラフが成立すると考えています。



●図 2.1　学習コストとアーキテクチャとスケジュール

「**制限が強いが学習コストが高いアーキテクチャ**」はチームメンバーの習熟に時間がかかりますが時間が経ち、アーキテクチャが浸透すれば高い生産性を発揮します。逆に「**制限が緩いが学習コストは低いアーキテクチャ**」は、チームメンバーの習熟に時間はかかりませんが、アーキテクチャに適切な制限を設けられていないことが多く、時間とともにコードが複雑化して生産性は下がります。

開発にあたって「コストパフォーマンスの見合ったアーキテクチャなのか？」と問われることがよくあります。つまり制限が強すぎるファットなアーキテクチャ選定にならないかという不安からくる質問です。

制限が強いアーキテクチャを選択してもよいケースの回答は次のとおりです。執筆時点のモバイルアプリ開発での筆者の肌感は、リードエンジニアがフルコミットできて開発期間が 4 ヶ月以上であれば、たとえリード以外がジュニアなメンバーだったとしても十分にペイしうるという感覚で

す。もちろん、新規開発の背景にも依存します。もしリリース後にアプリのメンテをしないと決定している場合は、学習コストは払わずに制限が緩いアーキテクチャを選択する方が開発期間を短くできるでしょう。

気にすべきポイントは、アーキテクチャがチームメンバーのモチベーションを下げうるかという視点です。この視点に対して「技術よりもプロダクトである」と回答できますし、それは至極真っ当な意見だと考えています。

しかし、モチベーションを軽視すべきではありません。あまりにレガシーでありチームメンバーのキャリアにとって何の役にも立たないような技術選定をしてしまうと退職リスクが高くなり、開発しようにも人がいないという状況になる可能性があります。そして、そういったプロジェクトでは、成長を期待できないことから当然採用もうまくは行きません。アーキテクチャはチームメンバーにとって学びになり、また採用にも効く技術選定をするとよいでしょう。

2.2.2 締め切り

チームメンバーの次に技術選定に影響を与える要素は**締め切り**です。締め切りのないプロジェクトの経験がある読者は、少人数ではないでしょうか。少なくとも筆者は締め切りと無縁であったプロジェクトの経験はありません。

前述の図 2.1 でも示したとおり、適切な制限を設けたアーキテクチャがチームに浸透するには少なからず時間がかかります。２つのパラメータ、つまりチームメンバーの習熟度と締め切りを照らし合わせてアーキテクチャを決めていくことになります。

ここに見落としやすいポイントがあります。当たり前のことですがリリース後も開発は続きます。「ここはリリース後に直すから」という魔法の言葉を多用して締め切りを優先してしまうと、その後の開発で大きな遅れが発生する可能性がでてきます。将来の生産性を犠牲にしていることには中々気づけません。

突貫で締め切りに間に合うよう作ってしまったために、その後の数年で負債を返済することに苦しむという話はどこでも聞く話です。プロジェクトの締め切りや、開発のマイルストーンを検討する場に見積もりのできるエンジニアがいることが理想ですが、そうではない場合が多くあるのも事実です。それでも締め切りに無理があるときは決裁者としっかり交渉をするようにしましょう。

ただし、交渉する前に締め切りの意味をかならず理解してください。なぜその締め切りが設定されているのか。どういう計画が予定されているのかはおさえておくべきでしょう。多くの場合、事業計画書などを眺めれば企画側（筆者は、プロジェクトを立場で分断する表現は好きではありませんが、ここでは企画者）がリリース後に予定している計画がわかります。

もちろん企画者に事情を聞くことも大事です。たとえば事業計画が新規アプリの最初のリリースでは基本機能のみを、次に目玉機能を作って、その目玉機能のリリース後に大きなマーケティング施策を実施するといった計画であれば、最初のリリースよりも目玉機能のリリースに焦点をあてる計画になるはずです。

これらを踏まえて交渉すると「新規リリースでは遅れの心配はないが、その場合、目玉機能のリ

リースで高確率に遅れてしまいそうです。新規リリースを少し後ろにおくことになるが、いま開発基盤を固める時間をもらえれば目玉機能のリリースは間に合いそうです、こちらでどうですか？」といったような交渉ができれば、締め切りというパラメータが調整できるかもしれません。

締め切りの問題は技術選定のボトルネックになりやすい要素ですが開発体制も事業計画もWin-Win になるスケジュールを設定できるように努めましょう。

2.2.3 主流を追う

ここまで紹介したチームメンバーや締め切りは、どちらかというと組織内部に起因するものでした。**主流を追う**とは、より広範囲の業界の技術トレンドに起因する要素といえます。その時々の主流な技術を取り入れることは次の視点より大事な要素と考えています。

主流な技術とは、まだどこにも導入実績のない最先端のテクノロジーではありません。プラットフォーマーが推奨していたり、ある程度モダンであるとの共通認識が確立されており、業界において多くのプロダクトで取り入れている技術のことです。

主流な技術を取り入れておけば、チームメンバーの参入障壁を低く保てるだけでなく、技術選定自体がエンジニアの採用に効く場合もあります。逆に主流から大きく外れてた技術選定をしてしまうとチームメンバーとなる参入障壁が高く、そもそもスキルセットがマッチせずに採用が難しくなります。

主流な技術であれば、すでにある程度の共通認識や習熟を想定できるためチームメンバーの学習コストも抑えられます。これらの理由から主流な技術を積極的に取り入れたアーキテクチャ選択や技術選定を心がけてください。

さて「Android アプリ開発における主流な技術とは何か？」という疑問に答えるのは、やはりAndroid OS の開発元である Google です。主流を追いかけるには公式ドキュメントや Google の技術発信が頼りです。プラットフォームだけでなくライブラリなども可能な限り最新のものをキャッチアップしましょう。

執筆時点では、年に一度の頻度で Android OS のバージョンが上がっています。その都度 Android SDK の API が追加されたり非推奨になったりとトレンドに合わせた変更があります。しっかりと新しい機能に対応し続ける必要があります。アプリが利用する Android SDK を更新していなければ Google Play Store でアプリを配信できない状況も起こりえます。時間をかけてゼロから開発した瞬間に負債を生み出していたという悲しいことにならないためにもドメインの最新技術にアンテナを張っておきましょう。

これは Android に限りませんが主流な技術というものは数年単位で移り変わります。非同期処理を実装する際、Android 開発においては過去、次のような変遷がありました。

1. Thread と Handler を使った原始的な実装
2. AsyncTask や Loader を使って非同期タスクに注目した実装
3. Thread と Event Bus を使ったイベント発生に注目した実装

4. RxJava などのストリームライブラリを使ったシーケンスに注目した実装
5. Kotlin Coroutines を使って Kotlin 言語とリソースに最適化した実装

執筆時点の最新の Android SDK では AsyncTask を利用した方法は**非推奨**（Deprecated）になっていますし、Java 言語のライブラリである EventBus を導入する事例も Kotlin の普及に伴って見当たらなくなりました。Android OS の進化がある以上は Deprecated API への対応が避けられません。アプリが配信できなくなるリスクを避けるためにも主流な技術をしっかり追いましょう。Android SDK の最新 API や Deprecated API の置き換えを手助けして互換性を維持してくれる Jetpack ライブラリなど積極的に使ってください。

そして主流な技術は時間とともに変わるものです。アーキテクチャと技術選定にあたってプラットフォームやライブラリへの依存が大きくなってしまうことは仕方がないことですが、変わることを前提とした柔軟性、変更しやすさも視野にいれてください。

2.2.4 会社のルール

組織の中では、技術選定に特定の縛りがある場合があります。ライブラリやアーキテクチャに対して社内標準を定めているケース等ですが基本的には従うしか方法がありません。

技術的な**会社のルール**はチーム間のメンバー流動化やスキルセットの平準化などを目的としていることが多いですが、何らかの理由で技術選定が Android アプリ開発の主流な技術から大きく外れていると、従うことそのものがプロダクトの利益を低減させます。

社内標準のアップデートが追いついておらず非推奨な API を使ってしまっている場合、または社内でフォークして改造していたが元のライブラリが開発終了しておりすぐに大規模な書き換えが必要だとわかっている場合などは顕著です。

そのままアプリを作ったとしても大規模な変更が目に見えている状態は会社としても、決してよいとはいえません。会社のルールそのものを変えることが難しくても取り組みの評価やフィードバックを前提に、実験的な許可を取り付けてもよいでしょう。技術の専門家という立場から交渉して技術選定の縛りを変える動きをすることをお勧めします。

2.2.5 Web API

サーバーサイドがどのような API を提供するかによって、使用するライブラリが自ずと決まってきます。

通信ライブラリでいうと REST である場合は Retrofit ライブラリの使用が主流です。ただし Kotlin MPP などに対応をしている場合は Ktor を使ってもいいでしょう。Web API が GraphQL で提供されている場合は、Apollo ライブラリを使います。もし gRPC であれば gRPC-java や gRPC-kotlin ライブラリを使うことがほとんどです。

また Web API の設計思想がアプリのアーキテクチャに大きく影響することもあります。サーバーサイドによっては「とあるコンポーネントの表示する際に API を叩く回数が 1 回だけである」

という期待値の場合もあります。この「コンポーネントを表示する際に API を叩く回数が 1 回」というのはデザイン上または性能上の制約であることが多く、その場合、これを満たすアーキテクチャを考える必要がでてきます。

サーバーサイドとクライアントサイドはチームが違ったりアーキテクチャが違ったりと明確な境界線がありますが、通信技術に何をどのように使用するかという点において、少なからず相互のアーキテクチャへ影響しあっています。

2.3 アーキテクチャを決める

背景を把握できたらアーキテクチャを決めていきましょう。抑えるべきポイントを粒度の大きい順に列挙してみます。

- アーキテクチャパターン
- コンテキスト分割
- 多層アーキテクチャでの分割
- データフロー
- 非同期
- 永続化
- エラーハンドリング

これらをしっかり決めることができれば、適切な制限を設定できたといえるでしょう。各ポイントの説明に加えて、筆者が設定する場合の例をそれぞれ説明していきます。あくまで例ですので「これが正しい」「これにしなければならない」ということはありません。各々自チームの参考にするなり、カスタマイズするなり、取り入れてもらえると幸いです。

2.3.1 アーキテクチャパターン

モバイル系で適用されるアーキテクチャパターンをざっと挙げてみます。すべてを網羅しているわけではありませんが次のアーキテクチャパターンが認知されています。

- MVC
- MVP
- MVVM
- MVI
- VIPER
- クリーンアーキテクチャ
- Redux
- Flux

第 3 章

アーキテクチャの浸透と改善

釘宮 愼之介／@kgmyshin

アーキテクチャには学習コストがあり、チーム全体に浸透してはじめてチームの生産性への寄与が最大化することを前提に本章では、この学習コストをより早く支払う方法、つまりチームに浸透しきるまでの時間を早める方法について説明します。

3.1 アーキテクチャが浸透している状態とは

アーキテクチャが浸透しているという状態の理想は、特定の機能を実装するとしたとき誰が書いても同様のコードが書かれることです。そこにメンバーのスキル差があったとしてもです。

このとき同様のコードになるとは一体どこまで揃っていればいいのでしょうか。次のようにいろいろなレベルに分けて考えられます。

1. 実装するクラスの分け方と名前が揃っている
2. 公開するインタフェースレベルと名前が揃っている
3. プライベートな関数やメンバーまで揃っている
4. 関数の中身の実装まで揃っている

チームメンバーの誰が書いてもこれらのうち（2）程度のレベルまで統一されていればアーキテクチャとしては十分に浸透しているといえるでしょう。（3）以降のプライベートな関数や中身の実装方法は、メンバーによって多少書き方が異なっても問題になることはありませんし、統一性にも大きな影響は与えません。例外として DataBinding のインスタンスの初期化方法や **DI**（Dependency Injection）の表記など、どの機能を実装するうえでも共通の事柄に関しては、実装方法まで揃っているべきだと筆者は考えています。

次に設定したアーキテクチャがこのチームに浸透しているのかを確認する方法を知っておきましょう。

前述の（1）および（2）が達成できていればアーキテクチャが浸透していると定義しました。この状態までメンバーが理解しているか確認する方法として、すぐに思いつくのはコードレビューです。コードレビューでは書き上がったコードのクラスとインタフェースが設定したアーキテクチャと隔たりがあるか確認できます。

コードレビューは、どこのプロジェクトでも行われている品質保証の仕組みです。アーキテクチャの浸透を確認するにあたって開発フローの中に特別に何か新しい手順を追加することなくできる良さがあります。ただ、このやり方ではコストが高くなることが多いです。開発初期はどうし

てもアーキテクトの思想や設計方針が伝わらず、乖離の大きいコードへのレビューとなるからです。その際に「全部書き直してほしい」と指摘をするのは、する側もされる側も、双方への心的ストレスを与えますし、そうでなくても二度手間で実装コストに跳ね返ってしまいます。またコードレビューは品質保証のためだけではなく、教育のために行うこともあります。教育の視点も含めた場合、指摘数はさらに増えてレビュワーのコストは跳ね上がってしまうでしょう。

筆者がお勧めするレビューはクラス図を書いてもらう方法です。クラス図であれば（2）の状態を理解しているかという観点をコードまで書かずに確認できます。クラス図は機能といったシステム構成をクラスの相互関係によって図示します。メンバーの開発習熟度によっては書いたことがない人もいるかもしれません。クラス図は視覚的に理解しやすい性質があることも手伝って、調べるなり教えるなりすればすぐに目的を達成するレベルで書けるようになります。アーキテクチャが浸透していないタイミングでは積極的にクラス図を活用しましょう。

3.2 アーキテクチャを浸透をさせるには

次にチームへのアーキテクチャの浸透を加速させるためにはどうしたらよいかを考えていきましょう。何の準備や工夫もないケースでは実装したコードに対して、テックリードが「これは違う」「これは合ってる」というだけでは長い時間がかかり、チームでのストレスもすごいことになるでしょう。コストとチームの負担を下げるための準備や工夫の仕方があるのでしょうか。本節では筆者が行っているアーキテクチャの浸透を早める方法を紹介していきます。

3.2.1 ドキュメントを用意する

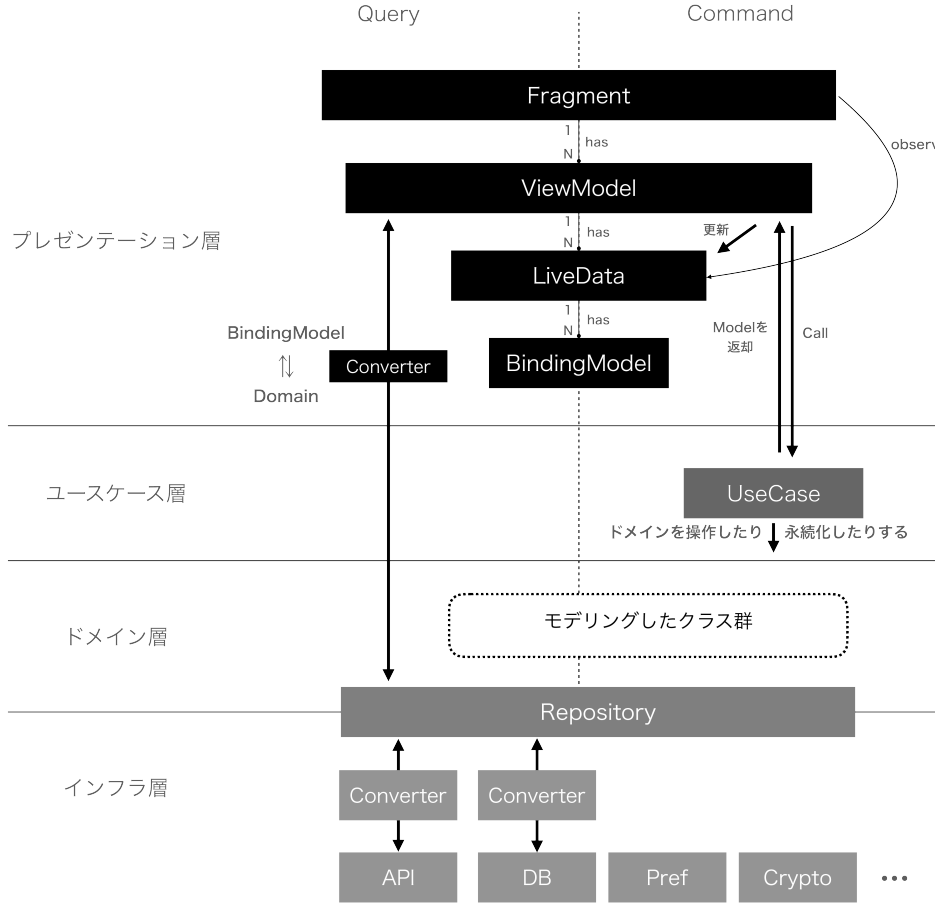
アーキテクチャのドキュメントは必ず用意しましょう。とはいっても、どういう粒度で用意すればよいのでしょうか。アーキテクチャの背景を含めてすべて書こうとしたら読みきれない分量になることも十分にありそうです。その分量を書いてもいいのですが、そのアーキテクチャに至るための経緯や知識をもれなく理解することがゴールではありません。目指すべきゴールは「チームメンバーがアーキテクチャに沿ったコードを書けるようになること」です。そのためドキュメントには「How」をメインで書くようにしましょう。理解を促進するために「Why」も書きますが分量は多くなり過ぎないように注意します。

筆者の場合マークダウンで 1 ページ程度で済む量を目安として順番に説明していく資料を作っています。

- 概要図
- パッケージ構造
- 層の説明
- 各クラスの説明

3.2.2 概要図

一目でどういうクラスを用意して、どういった層があるのか分かるような概要図を用意します。筆者の場合は Keynote や Google Slide などのプレゼンテーションツールで作成することが多いです。PlantUML など専用ツールもよいでしょう。筆者は図 3.1 のような図をよく概要図として用いています。



●図 3.1　概要図のサンプル

ところどころ、わかってほしい部分にコメントを入れて所有関係がどのようになっているかを書いています。この例では Command 側と Query 側とで層構造を変更しているアーキテクチャを採用しています。Query は ViewModel から直接 Repository を呼び出してほしいこと、Command は UseCase を経由しましょうということを示して必要なクラスを洗い出します。

3.2.3　パッケージ構造

各クラスのパッケージ構造が分かる図を用意します。プロジェクトのサンプルを作ってリスト3.1のようにtreeコマンド[注1)]で出力した結果をドキュメントに載せましょう。

●リスト 3.1　パッケージ構造を tree コマンドで出力する

```
com.kgmyshin.sample
    ├──di
    ├──data
    │   ├──repository
    │   │   └──TodoRepository
    │   └──api
    │       ├──ApiClient
    │       ├──ApiClientFactory
    │       └──json
    │           ├──common
    │           ├──request
    │           └──response
    ├──ui
    │   └──todolist
    │       ├──Converter
    │       ├──TodoListActivity
    │       ├──TodoListAdapter
    │       ├──TodoListFragment
    │       ├──TodoListBindingModel
    │       ├──TodoListViewModel
    │       └──TodoListViewHolder
    └──TodoApplication
```

プロジェクトファイルを開いても同様の構造をみれますがドキュメントでは対象を絞って理解できる点が強みです。パッケージ構造は機能や画面分割の基本となることが多いため、ここでフォローしておくとイレギュラー等にも気づきやすくなるためお勧めしています。

3.2.4　層の説明

採用したアーキテクチャが持つ層（レイヤー）を説明します。各層はどういう役割をもつのか、ここは層の概要に加えて属するクラスと配置の理由を主として個別の機能や特殊な事情は省略しましょう。

たとえばプレゼンテーション層のドキュメントは簡潔に次のようになります。

プレゼンテーション層（ドキュメントサンプル）

プレゼンテーション層はユーザーに情報を表示し、入力を受け付ける機能を持っています。

注1)　tree コマンドは Mac では標準搭載されていません。brew であれば brew install tree でインストールできます

プレゼンテーション層には主に Activity、Fragment、ViewModel、BindingModel に変換する Converter も配置します。View にかかわるものも、この層の役割です。リスト 3.2 のように RecyclerView を用いる場合の Adapter や ViewHolder などが対象です。

●リスト 3.2　プレゼンテーション層

```
com.kgmyshin.sample
    └──ui
       └──todolist
          ├──Converter
          ├──TodoListActivity
          ├──TodoListAdapter
          ├──TodoListBindingModel
          ├──TodoListFragment
          ├──TodoListViewModel
          └──TodoListViewHolder
```

3.2.5　各クラスの説明とサンプル

各クラスの説明は次の項目をそれぞれできれば十分です。

- 役割（責務）
- クラス名での命名規則
- ルール
- サンプル
- テストで確認すべきこと
- テストのサンプル

小さなクラスであれば理解すべき事柄も相対的に少なくなりますので必要に応じて省略できます。ここでは責務が大くなりがちな ViewModel に対してサンプルを用意しました。プロジェクト特有の事情があれば忘れずに記載してください。Android アプリでできることは数多くあるため新規に参画したメンバーは書きたい形でソースコードを書いてしまいます。チームが採用したアーキテクチャのもつ制約を把握できるようにします。

たとえば ViewModel のドキュメントは次のようになります。

3.2.6　ViewModel（ドキュメントサンプル）

責務

- イベント管理

第4章
より多くのチームへ

釘宮 愼之介／@kgmyshin

これまで新規開発の性質と選定すべきアーキテクチャに触れてきました。新規開発は多くのことをゼロから考えねばならず、またモダンな技術についてもキャッチアップが求められるタイミングでもあります。このタイミングはサービス開発において、特に**技術的知見が多く、深く蓄積できる**瞬間でもあるわけです。ここで得た知見を自チームだけに閉じ込めておくのは大変もったいないことです。会社で多くの事業、開発チームがある場合は、ぜひ**社内に共有**してみましょう。本章では自らが所属するチームだけでなく、いかにして会社全体の生産性向上を図るべきかに触れていきます。

4.1 社内にリポジトリを共有しよう

社内への知見共有で一番手っ取り早く行える方法は、社内へソースコードやドキュメントを共有することです。この話題は GitHub Enterprise 等を使用していたり、社員や関係者がひとつの Organization で管理されていたりと、ソースコードホスティングサービスが整っており**関係者のみ**がアクセスできる管理が前提です。

GitHub Enterprise を使用している場合でも種々の理由からプライベートリポジトリの運用が多くあります。事業ドメインによりソースコードを広く共有できないケースもあります。高い安全性を求められる事業、たとえば金融に関わる場合はこのような厳密な管理を行うとしばしば聞き及びます。

しかし、これまでの慣例に従ってプライベートリポジトリで運用しているといった不明確な理由に基づくケースも存在しています。機密情報や keystore ファイルをリポジトリ以外で管理しており、開発チームがセキュリティを担保できている場合、リポジトリの公開範囲を社内に広げることも考慮してみましょう。

共有するメリットについて説明しておきましょう。会社全体に向けて知見を共有できるという点が１つ目ですが、これ以外にもメリットはあります。

それは**新規のチームメンバーと同等の視点**での質問をもらえる点です。良いプロジェクトでは新規のチームメンバーが参画してからコードを書き始めるまでに必要な時間が短くなります。新規のチームメンバーがセットアップを終えて、コードを理解し、書き始めるまでに詰まるポイントをあらかじめ潰しておくことが望ましいわけです。ここでの課題はすでにチームに所属しているメンバーは記憶を消さない限りは真の意味で新規参加者の気持ちになることはできません。

社内でのソースコードとドキュメントの共有は他チームからのエンジニアから質問やアドバイス

をもらうきっかけになります。他チームのエンジニアは新規のチームメンバーと同様に自プロジェクトへの理解が浅く、彼らが抱く疑問は将来のチームメンバーの困るポイントの可能性が高いわけです。他チームのエンジニアから意見を集めることでオンボーディングコストの最適化を探っていけます。

デメリットも触れておきます。社内へリポジトリを公開するデメリットは、情報漏洩リスクです。基本的に会社で従業員として働いている限りは秘密保持契約、または同等のものを結んでいるはずなので契約で気にする必要はないでしょうが、リテラシーが不足している場合にはいくつかのリスクが存在しています。ありそうな想定リスクでは「プロジェクトの設定ファイルを参考に開発環境を更新し、バックアップのつもりで個人のリポジトリへ追加したところ、機密が含まれており、一緒に公開されてしまった」などです。リポジトリに含まれている機密は次のようなものがあります。

- アプリやサーバーの設定ファイル、環境設定
- サーバーや Web サービスの API キー
- テスト用のユーザー名とパスワード
- keystore などの秘密鍵ファイル
- テストサーバー等で利用する OAuth トークン

これらはファイルとしてもっていたり、ハードコードされていたりといろいろな形でリポジトリに含まれている可能性があります。社内でリポジトリを共有するまえにチェックしておくといいでしょう。

ほかにも社内の開発チームにどんな人がいるか分からない場合には共有するのもためらわれるものです。まずは社内でエンジニア同士の交流が生まれる場を作ってみてから徐々に共有することも考えていきましょう。

リスクを整理し、必要なステークホルダーに社内で共有する合意をとり、しっかり社内にアナウンスをしましょう。ひっそりと共有しても、だれも見てくれないので意味がありません。社内にチャットツールなどがあるのであれば、Android エンジニアが集まっているチャンネルがあるはずです。なければ作りましょう。そして、そこにこう書きこみましょう。「コードとドキュメントを社内に公開しました。うちのイケてるリポジトリを見て、みんなしっかり学んでくれ」と。あとは各所から飛んでくるであろうつっこみを待つだけです。

4.2 アーキテクチャ共有会を開こう

繰り返しになりますが筆者の主張は新規開発フェーズは特に技術的知見が貯まるフェーズであるということです。そして、どんなサービスにも新規開発というフェーズは存在します。知見を自チームだけでなく他のチームとも共有できるようになれば、会社としての技術情報の幅も広がります。

ただしリポジトリやドキュメントを公開しただけでは、どうしても深い意見がもらえません。普段の業務の中で隣のチームのコードを自発的に読むという人はなかなか居ないように感じています。**より深く、多くの意見をもらうために**有効な手段がアーキテクチャ共有会の開催です。勉強会という体裁を整えれば、普段の開発業務とは違ってその時間は集中できますし、社内に限ることで話せる事情や範囲も広くとれます。筆者の経験では共有会後に改めて自チームのリポジトリに注目が集まり、さまざまな質問やアドバイスを受けることが多かったアプローチです。

4.2.1 アーキテクチャ共有会のメリット

筆者が実施してきたアーキテクチャ共有会の取り組みについて説明します。メリットは主に３つあると考えています。

１つ目は各チームに眠っている暗黙的な知見を共有できることです。最先端のテクニックをどんどん取り入れて、技術的チャレンジをたくさんしているプロジェクトではモダンな知見がたまっていることでしょう。逆に社内には新規開発のフェーズを通り過ぎた歴史のあるプロジェクトもあるはずです。当初はどういう考えだったのか。もしリファクタリングをしてきたのであれば基準をどのように決めるのか、将来の予定があればその背景などを知れるでしょう。社内という状況では似た組織的背景を持っていることが多く、同じ土壌から生まれた別プロジェクトの選択は、自分たちのチームの決断にも役立つものです。

すぐに取り入れることができる知見はいうまでもなく重要であり、暗黙的な知見の共有がアーキテクチャ共有会の主目的といっていいでしょう。また直接、自プロジェクトで活かせない場合でも開発プロジェクトの考え方を知ることは貴重な経験です。エンジニアとしては将来役に立つものなので、しっかり耳を傾けましょう。

２つ目のメリットは交流ができる点です。普段からコミュニケーションがとれている会社は実はなかなかありません。このような技術的な知識をしっかり共有する場を設ける機会は技術向上に寄与します。顔や性格を知らない人に質問や相談ができる人はあまりいないでしょうが、それでも一度話してみる機会があれば以降はコミュニケーションしやすくなります。

３つ目は、ネットワークの構築です。２つ目と似ていますが他のチームのシニアエンジニアに相談ができることです。会社によっては開発チームによって力量のばらつきがあり、中にはシニアエンジニアが不在のチームもあります。入社してすぐという立場であれば技術面でも迷うでしょうし、バックエンドからフロントエンドへのコンバートであったりする場合、Android 特有の事情に疎くなりがちです。こういった場がなければ、相談しようにも社内の誰に相談できるのかもわかりませんし関係性を構築することもできません。開発チームに閉じた結果として、経験不足によってカオスなコードが生まれてしまうわけです。しかし勉強会や知識共有という立て付けがあれば困っている点や相談したい点として挙げ、社内のエンジニアから多くのアドバイスをもらえます。結果として会社全体の技術レベルが上がっていきます。

4.2.2 アーキテクチャ共有会の進め方

次にアーキテクチャ共有会の進め方について触れていきます。アジェンダは各チームごとに現状を発表し、質疑応答をして、相談したいところを議論するという手順を繰り返すだけです。発表資料には次のようなものがあるとよいでしょう。事前に準備してください。

- アーキテクチャ概要図（なければ構成が分かるクラス図や役割を記述したドキュメントを用意します）
- 使用ライブラリや CI/CD などの開発環境の資料
- それぞれの技術選定の理由や選定に至った背景
- 工夫しているところ・自慢したいところ
- 相談したいところ・困っているところ

アーキテクチャの振り返りにもなりますので勉強会を開いて損することはありません。早速、日時を調整して開催する旨のアナウンスを社内へ発信してみましょう。

4.3 推奨アーキテクチャを設定しよう

知見を共有して終わらせるのではなく、さらに会社全体で推奨するアーキテクチャを設定することも考えてみましょう。推奨アーキテクチャを設定する目的はいくつか考えられますがやはり新規開発時の**スピードアップ**ではないでしょうか。

ここまで読み進めてくれた読者は、アーキテクチャを設定するには多くのことを考えなければならないことを知っているはずです。推奨アーキテクチャを利用すれば多くの時間を省くことができ、新規開発の立ち上がりを早められるというわけです。メリットは他にもあります。それは最低限のコード**品質を担保できる**という点です。

事業領域がひとつのスタートアップはともかく、会社によっては複数のプロジェクトを抱えており、設計の経験や能力がまだないメンバーだけで構成される開発チームが存在します。この状態のままで進めると開発メンバーの意思に反してカオスなコードが生まれてしまいます。推奨アーキテクチャは、アーキテクトレベルのエンジニアがチームにいなくても最低限のコード品質を担保できるようにする安全装置です。

一方でデメリットもあります。サービスやチームに合わない可能性です。第 2 章で説明したように、アーキテクチャの選定は各プロジェクトの持つ背景によって異なります。たとえば多層アーキテクチャを採用するにしても、サービスの複雑さによって層の数を変更するという説明をしました。すべてのパターンごとに推奨アーキテクチャを用意するということは、なかなか難しいと思うので、ある程度複雑な場合を想定して考案することになります。それなりの課題を解決できる複雑さのアーキテクチャを考案する必要があるため、学習コストも比例します。そこまで複雑でもないサービスでカスタマイズせずに適用してしまってかえって生産性が悪くなる可能性があります。

もうひとつ言及しておきたいデメリットは開発メンバーに技術的知見がたまらないという点です。なぜ新規開発で技術的知見がたまるのかというと、それはアーキテクチャをゼロから考えることができる貴重な場であるからです。推奨アーキテクチャの利用はこの「考える」行為の省略を意味します。そのため技術的な発見が減ってしまいます。

4.3.1 推奨アーキテクチャ策定の判断基準

では推奨アーキテクチャの採用はどう判断すればよいでしょうか？これはアーキテクチャを構築できるレベルのエンジニア（アーキテクチャを構築してきた実績がある、またはそれに準ずる）エンジニアがどのくらいいるのか？で決めればよいと筆者は考えます。

開発チームごとに構築可能なエンジニアが十分な人数いるのであれば推奨アーキテクチャを用意する必要はありません。このケースでは先に挙げた技術的知見がたまらなくなるというデメリットの方が大きくなるという事情も設定しないという結論を補強します。技術面でリードできるエンジニアが各所にいるのであれば推奨アーキテクチャの有無はさほどスピードに影響はありません。アーキテクチャ共有会を開き、定期的な技術交換をお勧めします。

逆に、そういったことのできるエンジニアが少ない場合は、推奨アーキテクチャを設定しましょう。このケースではメリットをしっかり活かすことができます。また技術的知見がなかなか得られないというデメリットも薄れます。この点についてはコラムにて補足します。

Column. ジュニアメンバーもアーキテクチャを考えたほうが知見も深まるのでは？

何においても考えることは成長につながります。であればジュニアと呼ばれるエンジニアについても成長に繋げるためにアーキテクチャを構築してもらったほうがよいのではないかという議論があるかもしれません。

筆者はできあがった良いアーキテクチャを読み、それを参考に実装してもらう方がより早く成長につながると考えています。良いコードを書けるようになるには、良いコードを知っていなければなりません。つまり何をもって良いと評価するのかという観点をもっていることになります。アーキテクチャにもこれがいえます。良いアーキテクチャを知らず何が良いのかという観点をもっていないまま、ゼロベースで適切なアーキテクチャを構築するのは無理がありますし、できるとしても成長という観点では遠回りな方法です。

「守破離」という言葉があります。まずは徹底的に型を守って習得しよう。習得できれば徐々により良いものを得られ、型を破れるようになる。そして最後にはその型から離れて新しい型を構築できるというのが、この言葉に対する筆者の理解です。これはジュニアメンバーがアーキテクチャを構築できるまでの段階でもいえると考えています。

ジュニアと呼ばれるエンジニアは既存のアーキテクチャを参考に実装できるようになって徐々にその背景や制限の意味などを知るようになる（守）。それを続けているうちに、アーキテクチャの改善点を見つけるようになり修正を指摘したり Pull Request を送るようになる（破）。それらの経験を元にゼロベースでアーキテクチャを構築できるようになる（離）。

これがアーキテクチャを構築できるようになるための近道と筆者は考えています。

4.3.2　推奨アーキテクチャのドキュメントを作ろう

実際に推奨アーキテクチャの雛形をどう作っていくのか考えてみます。通常の設計資料と同じくソースコードとドキュメントで構成します。

まずは誰が作るかです。ケースバイケースですが横断組織であったり、アーキテクチャを構築できるレベルのエンジニアが社内で集まったりして構築するかたちがよいでしょう。誰にも利用されないという状況は避けなければなりません。完成前から社内にアナウンスしつつ利用するモチベーションを高めるような動きをしましょう。意見の集約を目的に構築す参加者を募る、お披露目会などで導入のためのきっかけを与えるのも手です。なるべく多くの Android エンジニアを巻き込むようにしましょう。

まずはドキュメントについてですが、筆者の場合は GitHub Pages を利用しています。第 3 章で紹介したようなドキュメントに加えて、アーキテクチャだけでなく CI/CD の設定やコーディング規約についても定めておきます。

Android アプリケーションのコーディング規約は、Google や JetBrains の出しているものを使用することがほとんどです。社内での独自ルールもあれば、ここに記述しておきましょう。あまりに独自基準が多いと導入コストが高くなるためお勧めしませんが必要だと感じるものは入れておくべきです。筆者の場合、リスト 4.1 の独自ルールを定めています。

●リスト 4.1　コーディング規約の追加例

````
## 返り型の記述を省略してはいけない

次のように返り値の記述を省略するのはやめましょう。

```
  // × bad
  fun getPosition() = {数字を返す処理}

  // ○ ok
  fun getPosition(): Int = {数字を返す処理}
```

理由は本来数字を返す処理のところを間違って文字列を返してしまった場合に、省略しない形で書いていれば必ずコンパイルエラーになります。しかし、省略してしまうとコンパイルエラーにならずに発覚が遅れるケースがあるからです。

```
  // × コンパイルエラーにならない
  fun getPosition() = {本当は数字を返す処理を書きたいが文字列を返した}

  // ○ コンパイルエラーになる
  fun getPosition(): Int = {本当は数字を返す処理を書きたいが文字列を返した}
```
````

このようなルールは各エンジニアの知見としてたまっているものです。該当プロジェクトだけでなく、共通利用の価値があると感じるルールをみつけたら設定していきましょう。

4.3.3 推奨アーキテクチャのサンプルプロジェクトを用意しよう

第 3 章でも説明しましたがアーキテクチャの理解には、実際に動かせるソースコードが必要です。サンプルプロジェクトの題材は、ある程度の複雑さを要します。Web API を利用して、ユーザーなどの登録機能がある TODO アプリなどです。単純な TODO アプリではない理由として GET メソッドだけでなく各種の HTTP メソッドが揃った API を使うことになること、ユーザー認証とリソースアクセスの権限の認可も必要になるため、標準的なアプリを実装するうえで十分に参考にできます。雛形としての役割を果たすはずです。

このような背景からサンプルではサーバーサイド側も準備しておくとベストです。リポジトリを clone したら、docker コマンドでサーバーが立ち上がり、Android Studio で Run すれば、サンプルアプリが通信して動き出すというのが目指すべき状態です。

とはいえ本書の読者は Android エンジニアがほとんどでしょう。そこで、今回紹介する TODO アプリの Web API を提供するデモ用のサーバーのコードを用意[注1)]しました。

■ https://github.com/kgmyshin/todo-server-sample

試す場合、こちらのサーバーを使ってサンプルプロジェクトを用意してみてください。Web API は次のものを用意しています。

●表 4.1　デモで用意している API 一覧

HTTP メソッド	URL	説明
POST	/auth/signup	ユーザーの登録
POST	/auth/signin	ユーザーのログイン
POST	/auth/refresh	トークンのリフレッシュ
GET	/tasks	タスク一覧取得
POST	/tasks	タスク追加
GET	/tasks/{id}	タスク取得
PUT	/tasks/{id}	タスク編集
DELETE	/tasks/{id}	タスク削除

README.md にも記述してありますが./gradlew modules:http-adapter:bootRun コマンドを実行すると http://localhost:8080 で動くようになります。あとは Android アプリ側のサンプルプロジェクトを実装するだけです。

ちなみに Android のエミュレーターから PC の http://localhost:8080 にアクセスする際は http://localhost:8080 や http://127.0.0.1:8080 ではアクセスできません。http://10.0.2.2:8080 を指定します。注意してください。繰り返しになりますが、あくまでデモ用途のものですのでエラーハンドリングなどがなく実務に耐えうるものではありません。サーバーサイドの実装の参考にはしないでください。

注1)　あくまでデモの用途のためエラーハンドリングなど細かい点は未実装です。サーバーサイド実装の参考にはしないでください

第 5 章

大規模なチーム開発へのオンボーディング

横幕 圭真／@KeithYokoma

本章は**大規模なチーム開発**（多人数で同時並行に複数の開発）が進むような大規模なプロジェクトのチームメンバーとして新たに参加する場面を想定しています。そのような組織に特有の課題やチームのもつ技術的課題を乗り越えていくための足がかりを得ます。

また大規模なチーム開発のための組織を俯瞰するとともに組織へ新しいメンバーを受け入れていく技術的オンボーディングについても解説します。

5.1 多人数で同時並行に複数の開発案件が進む組織

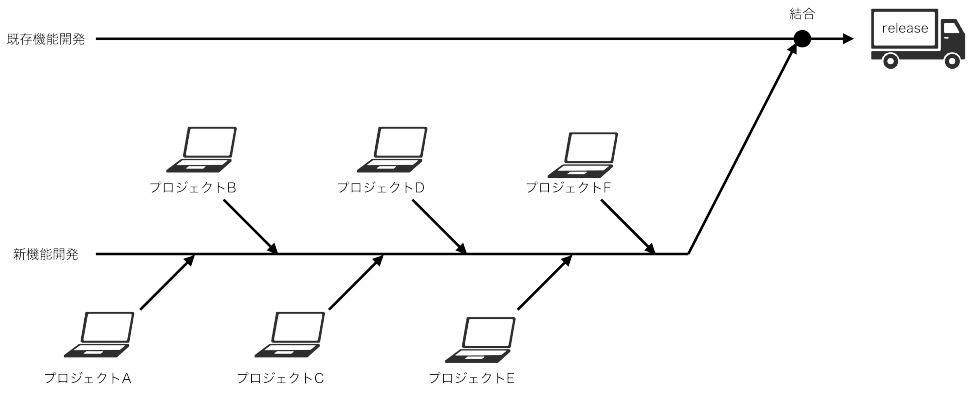
はじめに多人数で同時並行に複数の開発案件が進む組織がどのような姿をしているのかを整理します。

執筆時点では、日本国内のモバイルアプリ開発は大規模開発そのものの事例が少なく、まだまだ手探りの状況です。ここでは筆者が実際に開発に参加した、既存アプリへ新しく金融系サービスを組み込むために立ち上がった組織を例にあげて存在する課題や解決のための施策を紹介していきます。

5.1.1 新機能の組み込みを担う新たな組織体

筆者が開発に参加した組織では、新機能開発が始まる前からベースとなるアプリが存在していました。このアプリは最初のリリースからすでに４～５年ほどが経過しており、筆者が参加した時点で 10 人ほどの Android エンジニアが在籍して日々、機能開発していました。リリースのスケジュールも決まっており、およそ２週間に１度のペースで新しいバージョンをリリースできるよう企画・開発・QA の調整をしていました。

開発のワークフローとしては次の図 5.1 に示すように、既存機能開発と新機能開発をするチームがそれぞれに存在し、新機能開発の成果を既存機能に結合してリリースする戦略で進めています。



●図 5.1　新機能開発ワークフローの概略

ひとつのアプリに対し既存機能開発のチームと新機能開発をするチームの２つにわかれた理由は、あらたなサービスでは主に金融系の機能として、二次元コードやバーコードによる決済、NFC チップを介した非接触型決済、オンライン決済など多岐にわたる機能や要件があり、すでに開発していたアプリケーションとは独立した部分も多いことから、モバイルアプリチームだけでなくバックエンドチームも含めて新規に組織を作ることになった経緯からきています。

筆者が参画した時点で開発を始めてから１年弱が経過しており、最初のリリースを目指して６～７人ほどのエンジニアチームで開発の追い込みをかけていました。

開発組織がわかれたことで新機能開発のためのアーキテクチャも、これまでとは異なるものを採用することになりました。当初はリポジトリも分け、完全に独立したプロジェクトとして開発を進め、ひとつのライブラリとして既存アプリへ組み込むフローをとっていました。

こうして、ひとつのアプリケーションのなかに複数のアーキテクチャが混在することになりました。

5.1.2　マイクロサービス化されたバックエンドチームとモバイルアプリチーム

筆者の所属する組織では、提供するサービスがもつ機能をひととおり作りきってリリースすることを当初の目標におき、それをできる限り素早く開発するために複数の開発プロジェクトが同時に走っていました。

たとえば二次元コードやバーコードによる決済の機能開発を担うプロジェクト、NFC チップを介した非接触型決済を開発するプロジェクト、オンライン決済のシステムを開発するプロジェクトなど、機能単位のプロジェクトと、どのプロジェクトでも共通で利用する基盤機能を開発するためのプロジェクトがあります。

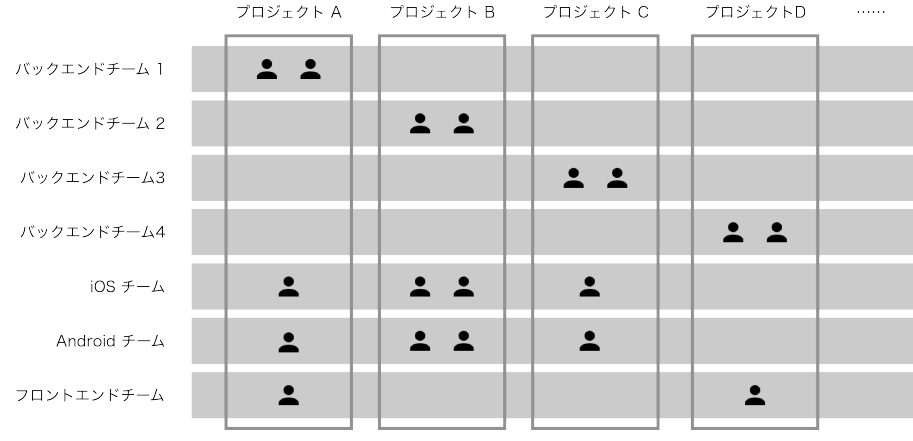
これらのプロジェクトが独立して機能開発を進めつつ、できるだけ素早くサービスをリリースす

るには、プロジェクト間の協調だけでは困難です。技術的にもうまく関心を分離し、機能同士の結合を、できるだけ疎に保つ必要がありました。

このため、バックエンドチームはマイクロサービスアーキテクチャを採用し、機能ごとにチームも分割されました。一方のモバイルアプリでも、エンジニアごとに別々のプロジェクトを担当しバックエンドチームと協力しながら機能開発を進める体制をとりました。これによってオンライン決済のシステムであるバックエンドとアプリの両輪を集中的に開発できるようになりました。

ここまでの事情を踏まえて組織にどのようなチームがあるかを紹介します。

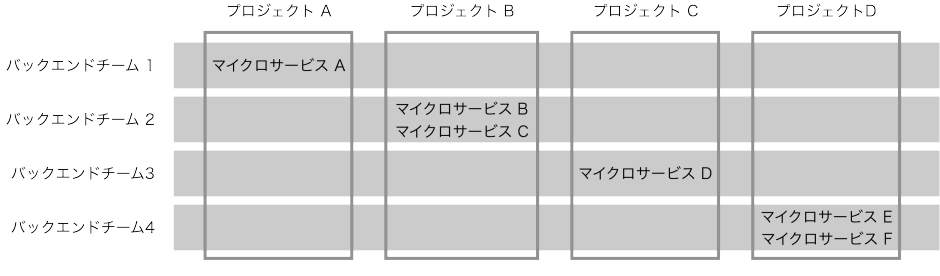
あらたなサービスの開発を担う組織の構成について概略図を図 5.2 に示します。



●図 5.2　マトリクス型の組織構成

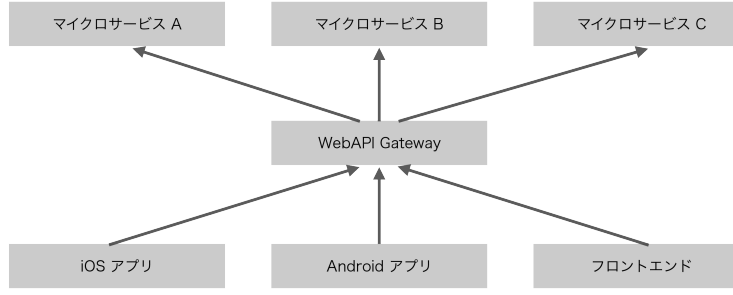
エンジニアリング領域による組織の分割（バックエンド、iOS、Android、フロントエンド）と、プロジェクト・機能別の分割があり、エンジニアリング領域によってわかれたチームから数人のメンバーが集まり、ひとつの機能開発プロジェクトを担当するマトリクス型の構成です。

バックエンドチームはプロジェクトごとに担当するサービスをもつ形をとっています（図 5.3）。



●図 5.3　バックエンドのマイクロサービスアーキテクチャと組織構成

モバイルアプリチームはプラットフォームによる分割（iOS、Android、フロントエンド）のみでチーム形成し、各メンバーは担当する機能開発プロジェクトごとにわかれる構成としました（図5.4）。



●図 5.4　単一の Web API を介したアプリとバックエンドの関係性

モバイルアプリチームとバックエンドチームの構成の差は、すべてのマイクロサービスとの連携のために単一の Web API を経由すること、モバイルアプリ特有の事情として最終的にひとつのアプリで配布するため、バックエンドを模したマイクロサービス化が難しいことに起因しています。

5.1.3　一貫したモバイルアプリのアーキテクチャ

モバイルアプリチームでは多くのメンバーが在籍し別々の機能を同時並行で開発していくため、モバイルアプリのアーキテクチャには次のような特徴が求められます。

1. 自分の開発が他のメンバーの開発に影響を与えない構成
2. どのメンバーも効率的に機能を実装できる共通のアーキテクチャ

さまざまな機能開発を同時に進めていくということは、コードに対し同時に多くの変更を加えていくということです。コードを変更する箇所が重複したり一箇所の変更が広範囲に影響を与えたりすると、コンフリクトした変更の修正や予定外の追加作業が発生し、その分だけ見積もりを超過する時間がかかってしまいます。

とくに多数の機能を開発する大きなプロジェクトでは、開発スピードとコードの品質を一定以上に保つ努力が求められます。簡単に作業を見積もることができる点も重要です。このため開発する機能ごとにコードを分割し、使い勝手のよいインタフェースで各機能を結合する構成が必要です。

機能を開発するプロジェクト間で担当エンジニアが流動的に変わる組織体制も影響しています。どの機能を開発するときでも、おなじ考え方やフレームワークを使えるようにし、担当するプロジェクトが変わっても開発スピードを落とさずに機能開発に集中できることも重要でした。

ただしモバイルアプリ特有の制約として、バックエンドにおけるマイクロサービスのようにイン

フラまで含めて機能ごとに分離できません。結果的にモノリシックあるいはひとつのリポジトリに複数のモジュールを配置する**モジュラーモノリス**なアーキテクチャにならざるを得ないという制約もあります。

筆者の参加したプロジェクトでは新しいサービスを最終的にひとつのアプリとして既存のアプリに組み込んでリリースするため、開発に用いる言語やフレームワークの管理を機能ごとに独立して行うバックエンドチームのようなマイクロサービスアーキテクチャをそのまま適用できませんでした。

それでも、マイクロサービスアーキテクチャのように機能ごとの結合を疎に保ち互いの変更による影響を最小化する手法は有効です。モバイルアプリ開発でも実現するため、ひとつのリポジトリの中でモジュールを細かく分割するモジュラーモノリスという形式を選択しています。

機能や画面の実装に用いるアーキテクチャについても、どのモジュールでも共通して適用できるアーキテクチャのパターンを模索しました。筆者が開発に参加した組織では、次のような理由でMVVM と Redux を組み合わせたアーキテクチャを採用しました。

1. アプリケーションの中で複雑に入り組みがちなデータフローを単方向に整えたい
2. ひとつの画面で取り扱う状態をうまくオブジェクトで表現したい

Redux はもともとウェブフロントエンドで使われた設計のパターンで、アプリケーションがもつ状態をひとつのオブジェクトで表現し、そのオブジェクトの状態を変更する関数を作って呼び出していくことで状態遷移を実現するものです。

このような背景をもつ組織では、Redux の特徴がアプリケーションの挙動のわかりやすさを助け、重要な要素であるビジネスロジックを関数として表現し簡単にテストが記述できるとの期待から Redux を採用することにしました。

また開発対象のアプリケーションでは、ひとつの画面にさまざまな要素を表示するため多くの状態を定義する必要があったり、複雑な状態遷移をする機能があったりしたため、Redux の設計パターンを画面ごとに当てはめて使いました（これはどちらかというと Flux のパターンに合致しているかもしれません）。

結果としてマルチモジュールによるモジュラーモノリスな構成にたどり着きましたが、この決断の背景には新しい金融系サービスには多くの機能があり、それぞれの機能開発を分離して変更の影響を最低限に留めるという、ドメインや組織といった大規模開発のバックグラウンドが影響しています。

モジュラーモノリスの構成では、画面の実装を持つすべてのモジュールで MVVM と Redux を組み合わせたアーキテクチャを利用することで、担当プロジェクトが変わっても同じ考え方を使って機能開発ができること、異なるモジュールの機能開発でもコードレビューがしやすくなることを目指しています。

5.2 大規模開発へのアプローチの一般化

ここまでで筆者の所属した組織についての背景から、どのような考え方から MVVM と Redux を組み合わせたアーキテクチャを採用したのかを解説しました。

本節では、これまで解説してきた大規模開発へのアプローチの一般化を試み、大規模開発をうまく進めていくための要点を整理します。

5.2.1 大規模開発の進め方

筆者の所属した組織では、ビジネスの要件として新しいサービスを既存のアプリにアドオンする形で提供するという意思決定があったため、既存アプリ開発チームと新しいサービスの開発チームのそれぞれで独立して開発を進める大規模開発の方法をとりました。

別の選択として一気に新規アプリを開発し、最初のリリース時点で多様な要件を満たすためあらゆる機能をすべて実装していく形式の大規模開発の進め方も考えられます。

新しいサービスを既存のアプリに組み込んでいく方法と、一気に新規アプリを開発していく方法で、大規模開発の進め方に、なにか違いは生まれるでしょうか。

新しいサービスを既存アプリに組み込んで進める場合には、新しいサービスで採用するアーキテクチャへの理解に加えて既存機能の実装方法やアーキテクチャについて一定の理解が求められます。

筆者の場合、ビジネス要件として既存機能に新しいサービスの機能を結合して利用することが求められたため、チームの枠を超えてお互いのアーキテクチャを理解し合い、実装を進める必要がありました。

一方、新規アプリであれば、開発チームのなかでどのようなアーキテクチャを採用するか合意をとっておくことで、少なくとも開発の初期段階で異なるアーキテクチャが同一アプリに存在することはなくなります。

いずれの場合でも共通していることは、開発チーム内でどのようなアーキテクチャを採用するか**合意のもと進める**必要があるということです。アーキテクチャの合意がないまま、チームメンバーそれぞれに得意なアーキテクチャを採用して開発をしていくことも不可能ではないかもしれませんが、実装箇所ごとに開発を担当した人にしかわからない属人化が起きたり、アーキテクチャそのものを理解するためのコミュニケーションと実装の意図を理解するためのコミュニケーションの両方が必要になったりします。

新しいチームメンバーを迎え入れることを想定すると、アーキテクチャの数だけ習熟しなければならないことが増えていきます。よって大規模開発に至るビジネス的な背景にかかわらず、アーキテクチャについての合意が大切だといえます。

またアーキテクチャは組織やビジネスの要件に紐付いて決まります。大規模開発では、これらの制約がより強くなります。

第 6 章

大規模なチーム開発とアーキテクチャ

横幕 圭真／@KeithYokoma

前章では大規模なチーム開発について組織構造の特徴を捉え、複数の機能開発を同時並行に進めるためのアーキテクチャの考え方について解説しました。本章ではその考え方をもとに実際の Android アプリプロジェクトに落とし込み、とくに Android アプリプロジェクト内におけるモジュラーモノリスの実現方法、具体的なアーキテクチャの実装例、最終的にひとつのアプリアプリケーションとして機能を結合する手法について理解を深めます。

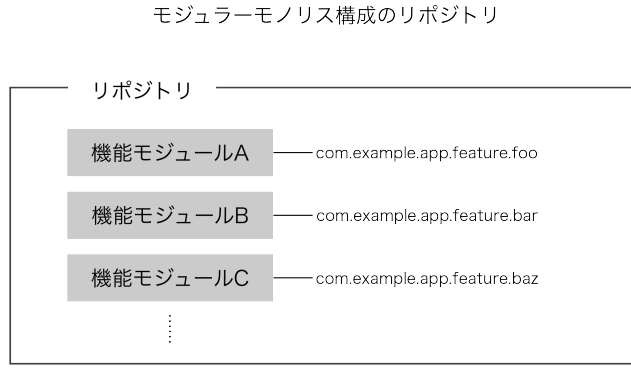
解説するアーキテクチャ事例は筆者の所属したチームで採用しているアーキテクチャの基礎をなしています。大規模開発を支えるために Redux の仕組みを用いた MVVM アーキテクチャで UI やビジネスロジックの実装と、簡易なレイヤードアーキテクチャを採用してモデル定義や API アクセスなどを抽象化したインフラストラクチャを分離し、定義しています。

さらに機能ドメインごとモジュールにまとめて関心を分離するモジュラーモノリスの構成をとっており、単一のリポジトリ内に数多くのモジュールを保持しています。重厚長大で複雑そうですが、チームメンバーが多数所属する筆者の組織で複数の機能を同時並行に開発する体制を支えるには、ある機能の開発が別の機能の開発へ与える影響度を極力減らす構造が必須でした。モジュールという単位で関心を分離するモジュラーモノリスはこのような大規模開発にフィットする構成です。

執筆時点で数十人を超える Android エンジニアが協調し、開発を進めている大規模運用と開発効率の両立に耐えた実績のある手法ですが完全ではありません。アーキテクチャは大規模なチーム開発に対しても重要な役割を果たします。その理解の土台として本章を活用してください。

6.1 モジュラーモノリスなプロジェクト構成

モジュラーモノリスなプロジェクト構成とは、ひとつのリポジトリのなかに複数のモジュールを配置し、それらのモジュールを結合することでひとつのアプリケーションを成立させる構成のことを指します。Android アプリのプロジェクトでは特に **Gradle** による強力なサポートのもと複数のモジュールをビルドし単一のアプリケーションを生成する仕組みを用いてモジュラーモノリスな構成を実現します（図 6.1）。



●図 6.1　モジュラーモノリス構成：数十以上のモジュール分割も珍しくない

Kotlin や Java のもつ言語機能としてのパッケージ単位で分割するだけでなくモジュール単位で分割することで、他のモジュールに公開するクラスやインタフェースを限定でき、疎結合な設計を保ちやすくなります。またモジュールの命名によってそのモジュールがどんなドメインを担当しているのかをわかりやすくしてくれます。

ここではマルチモジュールの利点を活かせるよう、うまくモジュールを分割していく方法と、分割したモジュールをどのようにして結合するのかについて解説します。

6.1.1　モジュール分割の考え方

ひとつのアプリケーションを複数のモジュールに分割する目的は、他のモジュールに公開するクラスやインタフェースを限定し疎結合な設計を保ちやすくすることと、モジュールの命名によってモジュールごとにどんなドメインを担当しているかを明確化することです。

特に多数の機能開発が同時並行に進む大規模なチーム開発においては、日々ソースコードに加えられる大量の変更が及ぼす影響範囲を極力狭くすることで機能の結合を可能にします。開発スピードを落とさず一定のペースでアプリのリリースを実現したいという要求が背景にあります。そしてこの目的にあったモジュールを作るには、一定の規則や考え方を整えておく必要があります。

モジュールの分割方法にはさまざまな手法が考えられます。筆者の担当したプロジェクトでは主に機能ドメインで区切る縦割りのモジュール分割と、横断的コンポーネントを持つ横割りのモジュール分割の 2 種類を組み合わせてモジュール分割を実現しています。

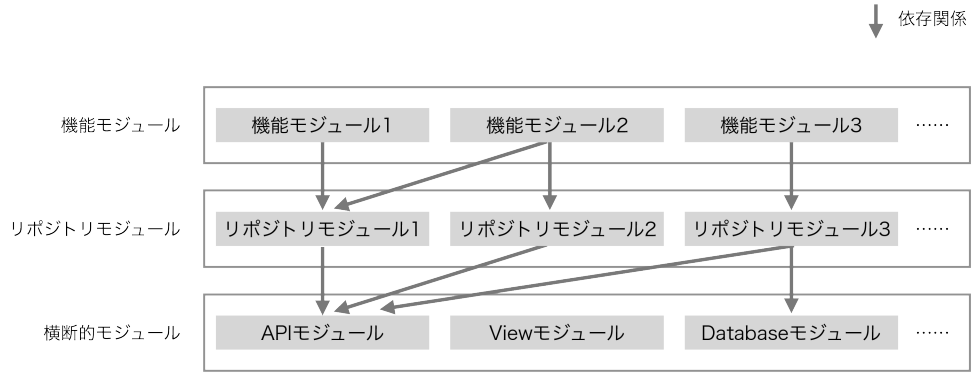
プロジェクトによっては、レイヤードアーキテクチャの考え方を使って UI 層のモジュール、アプリケーション層のモジュール、ビジネスロジック層のモジュール、インフラストラクチャ層のモジュールの４つのレイヤーごとにモジュールを分割する方法も考えられます。

どのモジュールの分割方法がチームにフィットしているかはチームやドメイン、プロジェクトの性質によって異なります。同時に進むプロジェクトの数やチームメンバーの人数、プロダクトその

ものの複雑性によって、レイヤードアーキテクチャの各レイヤーによる分割のみを採用し４つのモジュールを作れば十分な場合もあれば、さらにドメインごとの分割も加えてモジュールを細かく分割する必要がある場合もあります。

一度モジュール分割の方法を確立したあとでも、開発を進めていくにつれてプロダクトの成長に伴ってアプリの複雑度も上がっていきます。現状のモジュール分割の方法がチームにフィットしているか都度、振り返るようにしましょう。

ここでは筆者の担当したプロジェクトを事例にあげ、「機能ごとに区切られた機能モジュール」「モデルを定義し共有するためのリポジトリモジュール」「横断的コンポーネントを持つ共有モジュール」の３つのパターンを組み合わせたモジュール分割について、次の図 6.2 に示すような構成を生み出した背景を解説します。



●図 6.2　モジュール構成の作り方の例

6.1.2　機能ベースで区切る縦割りのモジュール分割

モジュールの分割方法のひとつとして、機能を単位としてモジュールを分割する方法があります。

筆者の担当したプロジェクトでは決済に関する各種機能を実装しており、バーコード決済機能をもつモジュール、オンライン決済機能をもつモジュール、各種設定をもつ画面を担当するモジュールなど、UI に関する実装をドメインごとに集めたモジュールを作っています（リスト 6.1）。以後このようなモジュールを機能モジュールと呼びます。

●リスト 6.1　機能ベースで分割したモジュールの例

```
* project root
|
+- * feature_barcode_payment: バーコード決済機能
+- * feature_charge         : チャージ機能
+- * feature_history        : 決済履歴機能
```

```
+- * feature_home            : さまざまな機能の入り口となる画面
+- * feature_kyc             : 本人確認機能
+- * feature_nfc_payment     : 非接触決済機能
+- * feature_online_payment : オンライン決済機能
+- * feature_settings        : 設定画面
+- ...
```

機能ごとにモジュールを分割する利点として、他の機能には必要のないクラスを隠蔽し結合に必要なインタフェースのみを公開できるようになります。これによって、他のモジュールに向けて利用してほしいインタフェースと利用してほしくないインタフェースを区別し、他のモジュールからは利用すべきインタフェースだけを意識すればよくなります。

Kotlin を開発言語とした場合、パッケージごとに機能をまとめたとしてもパッケージ内のみに公開する可視性修飾子が存在しない[注1)]ため、その機能でしか使うことのないクラスもすべてアクセス可能な状態になります。一方で Kotlin には同一モジュール内のみに公開するための可視性修飾子（`internal`）があり、機能ベースで区切るマルチモジュール構成と組み合わせると特定の機能にのみ公開するクラスを定義できるようになります。

このモジュール分割方法では、各機能で共通して利用するユーティリティやリソースの管理に課題が残ります。各モジュールで似たようなユーティリティを実装したりリソースを増やしたりしていくと、その分だけ重複するコードが増えていきメンテナンスの手間も増えてしまいます。

また画面遷移を伴う機能モジュール間の結合方法についても考慮が必要です。機能モジュール同士で依存関係を定義し、呼び出したい画面を直接参照する場合を考えてみましょう。画面遷移の方向が必ず一方通行になるならば問題ありませんが、いろいろな機能を行ったり来たりする画面遷移や、ある機能と別の機能がお互いに参照をもつような場合、簡単に循環参照が発生します。よって、機能ベースのモジュール同士で直接依存関係を持つのは得策とはいえません。

これらの課題の解決策については次節以降で解説をしていきます。

6.1.3 モデルを定義し共有するためのモジュール分割

機能を単位としたモジュールの分割では UI に関する実装をドメインごとに分割していました。このとき各機能で必要となるモデルを同じモジュール内で管理してしまうと機能モジュール同士の依存が生まれません。

一見、よい選択にみえますがこれでは横断的なモデルの共有ができなくなります。たとえばユーザー情報を持つモデルやプロフィール情報を持つモデルなど、どの機能でも共通で使いたいモデルが共有できていないと同じモデルの定義をあちこちのモジュールで作ることになってしまいます。

この問題を解決するには、次のリスト 6.2 のようにモデルの定義をするモジュールを別途用意し、各機能モジュールは必要とするモデルを持つモジュールへ依存する形式を取ります。命名はモデルを持つことが分かるか、データソースレイヤーであることを明示するような名前にします。

注1） Java 言語における可視性修飾子のうち、なにも使用しない場合の可視性に相当するものが Kotlin にはありません

●リスト 6.2　モデルを定義するモジュールの例

```
* project root
|
+- * repository_barcode_payment: バーコード決済モデル
+- * repository_history        : 決済履歴モデル
+- * repository_kyc            : 本人確認モデル
+- * repository_nfc_payment    : 非接触決済モデル
+- * repository_online_payment : オンライン決済モデル
+- * repository_settings       : 設定モデル
+- ...
```

6.1.4　横断的コンポーネントを持つ横割りのモジュール分割

機能モジュール以外に共通して利用するコンポーネントを保持するモジュール分割の考え方があります。このモジュールはユーティリティや HTTP 通信のベースとなる部分、共有して使うリソースなどを管理し、機能ごとのモジュールが使いたいリソースを選択して使うことを想定したモジュールです。

このようなモジュールを用意することで、機能モジュールそれぞれで同じユーティリティを実装したり、同じリソースを別名で保管したりといったソースコード上の重複を避けられるようになります。

共通コンポーネントをもつモジュールで起こりがちな問題として、あらゆる共通コンポーネントが一緒くたになってひとつのモジュールに保管され肥大化していく問題があります。

ひとくちに共通コンポーネントといっても、どのような種類の共通であるかは個別に異なります。たとえば HTTP 通信を取り扱う共通コンポーネントと共通で使うカスタム View の実装がひとつのモジュールにある場合、カスタム View の実装を使いたいモジュールに対し、HTTP 通信を取り扱う共通コンポーネントまで公開してしまいます。

利用する機能モジュールからみたとき、どの共通コンポーネントを利用するかを選べるようにしてください。横断的なコンポーネントの種類に応じてモジュールを分割します。これにより、ひとつのモジュールにあらゆるユーティリティや便利メソッドを混在してしまう事態を避けられます。

筆者の担当したプロジェクトでは、リスト 6.3 のように取り扱う問題領域に合わせた命名を用いて共通コンポーネントをもつモジュールを分割しました。

●リスト 6.3　共通コンポーネントをもつモジュールの例

```
* project root
|
+- * common_android   : Androidフレームワーク用のユーティリティ群
+- * common_navigation: 画面遷移のインタフェースを提供するモジュール
+- * common_network   : WebAPIを呼び出すためのクライアント実装
+- * common_proto     : Protocol Buffers用のユーティリティ群
+- * common_resource  : 共有リソース
+- * common_redux     : MVVMパターンにReduxを組み込むためのベース
+- * common_ui        : 共通UIコンポーネント
+- ...
```

第 7 章

プロダクト開発をしながら技術課題の改善もする

横幕 圭真／@KeithYokoma

前章では大規模なチーム開発において Android アプリのアーキテクチャがどのように実装されているかについて解説しました。モジュール間で一貫したアーキテクチャを採用し、どのモジュールでも共通の考え方でコードを理解し実装を進められるようにすることと、一度に複数の開発がスムーズに進められるよう関心の分離をすることを主目的としたアーキテクチャですが、開発が進むにつれて重複した実装が増えるような技術課題がみつかったり、アーキテクチャそのもののさらなる改善点がみつかったりします。

本章では、日々の開発案件を進めながらもアーキテクチャや技術課題の改善や修正をするためのプロセスについて解説します。大規模なチーム開発ではプロダクトを支えるアーキテクチャは、事業ドメインや組織構造に強く依存します。開発事例そのものも少ないため、ここで紹介する手法は貴重な事例となるでしょう。

一般に開発規模の増大は、開発者一人あたりの生産効率低下を引き起こします。モジュールやコードの相互依存や増していくコードの複雑度、開発要件の難易度、繰り返される改修、コミュニケーションコストなどが主要因となり、大規模なチーム開発では徐々に開発速度の低下を感じることになります。そのような状況下でも足を止めず、チームとして課題に取り組める状態をつくるため既存コードのもつ背景を探求し、機能開発と改善の両立手法を議論します。

7.1 技術課題の認識を揃え、チームとして取り組める状態をつくる

日々の機能開発を進めていくにつれ、技術課題も少なからず蓄積していきます。また技術課題といっても多種多様な種類があります。動作の変更を伴うことのない軽微な修正で済むものもあれば、すべてのプロジェクトで共通する部分を大胆に変更しなければならないもの、また開発者が普段使用している開発ツールに関連するもの、ひいてはプロジェクトや開発の進め方に関わる部分にも技術課題はあらわれます。

ここでは、そのような技術課題たちに対してチームとして向き合い、改善していくための足がかりを作ります。

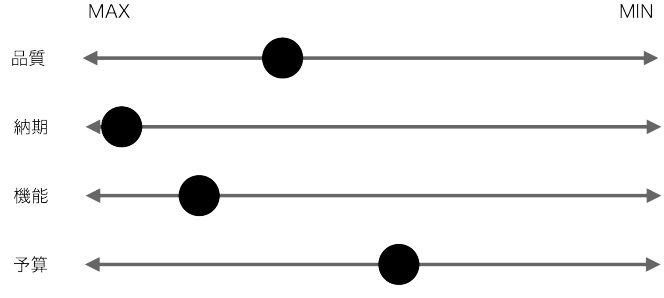
7.1.1　初回リリースまで全力で走り続けてきたプロジェクト

筆者の担当したプロジェクトでは、新機能の初回リリースへ向けてひたすら機能開発に集中していました。この新機能はアプリとバックエンドの連携だけでなく外部サービスとの連携が必要な機能が含まれたため、プロジェクト開始時からリリース目標が定まっていました。

このため初回リリースへ向けての開発では、Redux と MVVM を組み合わせた構成が最適であると判断し、アプリケーションレイヤーとリポジトリレイヤーを分離しています。アプリケーションレイヤーに位置するビジネスロジックを検証可能にすることを品質目標に定めて開発を進めていました。

新しいサービスを始める上で重要な機能群を素早く開発し、できる限りアプリの動作を安定させる状態でアプリをリリースするという要件は一見して過大と感じるかもしれません。

程度の差こそありますが、このような要件があるプロジェクトの表現方法に**トレードオフスライダー**があります。トレードオフスライダーは要件の基準を可視化する手法でアジャイルなプラクティスで用いられます。筆者が遭遇したリリース直前のプロジェクトの状態は次の図 7.1 のように表せます。

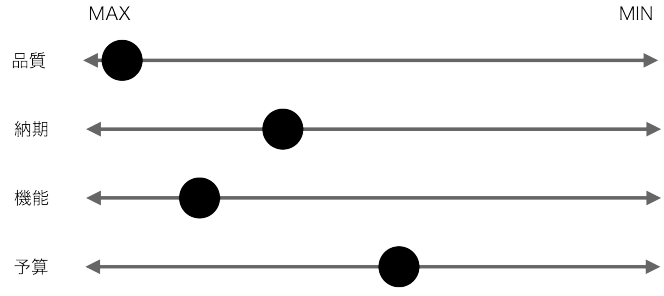


●図 7.1　リリース直前のプロジェクトへの期待値：納期と機能開発への集中

筆者の経験した大規模開発ではリリースまで１年以上を要しており、初回のリリース後もしばらくは機能リリースが継続していました。リリースが最優先となる状態が続くなか、チーム全員で機能開発に集中していましたが、新サービスで実現したい機能がそろってきた段階で変化が起きます。

機能開発にフォーカスする組織から機能を安定して運用し続けていく組織へと徐々に移り変わっていきました。初回リリースこそサービス開発の競争が激しい中でリリースを優先する体制をとっていましたが、新しい機能は金融系サービスであるという背景から、今後の開発を考えると「いつでも安定して機能が使える」ことへの要求が高まってきました。安定して動作するアプリを作るという品質を高めるための体制へと移行していきます。

リリース後のプロジェクトに対する要求は次の図 7.2 のように表現できます。



●図 7.2　リリース後のプロジェクトへの期待値：納期から品質へ優先度が変化

リリース日を最優先とした体制から品質を最優先とした体制へと移行したことがきっかけとなりチームで技術課題に取り組むことになったことは間違いありません。

しかし体制の移行以前からチームが抱える技術課題そのものは存在しており、チーム内でも解決をしたいという話題がしばしばみられました。リリース日を最優先とした場合、解決に割けるリソースと優先順位が低くなりがちであったため、チームはそこに技術課題があることを認識するにとどまっていました。

体制の移行を期にチームは技術課題に向き合い、改善していくことに注力します。改めてチームが抱える技術課題にはどんなものがあるかを整理し、新しいサービスを安定して動作させ、一定の品質を保った状態でアプリをリリースし続けていくチームへと変わっていく必要がありました。見つけた技術課題を改善するため、組織へ示していく取り組みを始めることになりました。

7.1.2　チームとして技術課題と向き合う

技術課題とは、そもそもどんなものを指すのでしょうか。

一言でまとめるなら、技術課題とは**日々の開発をすすめる上で障害となってしまうできごと**[注1)]です。技術課題があることがボトルネックとなって開発スピードが低下したり、本来必要のない作業の無駄が発生して時間を余分にかけてしまったり、コードの品質を低下させて意図しないバグを生み出したり多くの弊害を引き起こします。

品質の技術課題をとっても直接的なものでは検証にかかる時間を増大させるものや不具合の発生率を上げるもの、間接的なものでは生産性を低下させることで開発チームへ影響を与えます。組織体制などチームが持つ背景も考えると多様な技術課題が存在しますが、問題そのものを認知しなけ

注1)　広くチームが抱える課題を意味していますが本書では特にアーキテクチャに重点を置いて解説しています

れば理想的な解決は見込めません。

- 直接的な技術課題
  - 開発者に影響するもの：テストが書けない、手作業の動作検証が大変など
  - 組織に影響が波及するもの：ユニットテストのカバレッジが不足して不具合の発見が遅れるなど
- 間接的な技術課題
  - 生産性に影響するもの：テストの実行時間がかかる、コードの静的解析に欠陥があるなどワークフローの不備

ユニットテストが書きづらく、テストケースが動作すべてを検証しきれない場合、変更のたびに開発者の手で動作確認をしなければなりません。本来ならユニットテストによって自動で検証したいところです。技術課題があるばかりに、その分の工数を多く見積もることになります。

また想定外の動作やバグを作り込んだとしても、ユニットテストがなければ問題の発見が遅れる可能性もあります。検証担当者の手によって見つけられればまだいいですが、リリース後に発覚すると手戻りの影響は甚大です。これはテストのカバレッジが不十分なこと、あるいはユニットテストが書きづらくなっている設計が遠因の技術課題であるといえます。

生産性に影響するものとしてはユニットテストやアプリのビルドを日常的に自動で実行する CI の実行時間が長過ぎるなどがあたります。必要以上の待ち時間が発生している、CI に静的解析がないことで自動化できるはずのコーディング規約違反や実装ミスを都度レビューしなければならないときは生産性向上のために開発環境を整える必要がでてきます。品質保証そのものではありませんが間接的な技術課題があるといえます。

筆者の担当したプロジェクトで整理したところ次に挙げるような技術課題がありました。

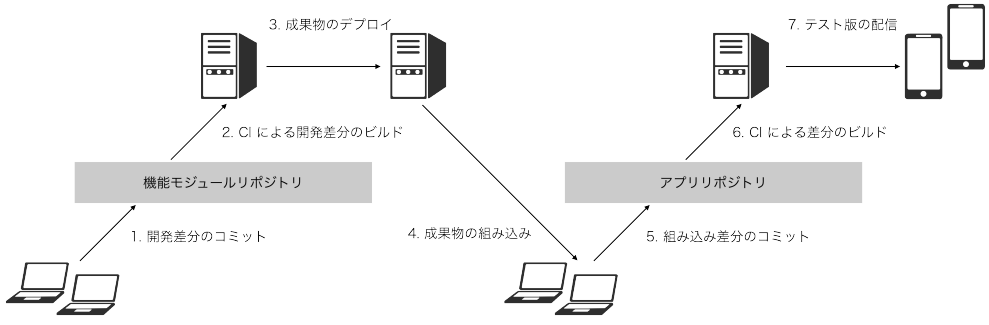
1. 頻繁に変更される UI の振る舞いをユニットテストで確認できていない
2. CI によるビルドの並列実行が設定できておらず、同時実行数にも強い制限があったためビルド待ちが多発していた
3. 基本となるフレームワークはあるがボイラープレートが多く、何度も重複した処理を書いていた

前述のとおり、技術課題は常にソースコードに問題があるとは限りません。チームメンバーが日常的に実行しているワークフローに課題があり、それがボトルネックとなってしまうケースも考えられます。

たとえば変更したコードをレビューするワークフローを導入していたとして、コードレビューの依頼に気づかれずに何時間や何日という単位でレビュー待ちになるような場合、コードレビューのワークフローに改善余地があります。特に同じコードベースを利用し、開発機能が多岐に渡る場合は、コードレビューを待つ間も主流のブランチにコミットが積み上がっていきます。

大規模開発では主流ブランチに毎月、数千コミット以上がマージされていきます。このような状況の中でコードレビュー待ちが発生すると、それだけで衝突の可能性も高くなってしまいます。

プロジェクトでの経験からコードレビューの待ち時間が長くなることや、それによるコンフリクトのリスク上昇のほか、ひとつの機能を実現するために２つのリポジトリにまたがった結合が必要でした（図 7.3）。



●図 7.3　２つのリポジトリにまたがった機能開発と結合のステップ

手元にあるマシンでのビルドと CI によるビルド、これを２つのリポジトリで実行しなければなりませんでした。結果として手元で開発中の機能を動かすビルドを作るステップはもちろん、QA 用のビルドやリリース用のビルドを作るステップも含めて、アプリの成果物を作り出すワークフロー全体が手間のかかる状態となっていました。

チームが拡大して新しいメンバーを迎え入れたり、退職や異動等でメンバーの構成が変わったりする頻度が高い場合、技術的知見がチーム全体に浸透していかなかったり、チームで持っている技術的知見に偏りができたりしてしまいます。

特に筆者の担当したプロジェクトの場合、チームメンバーは別々のプロジェクトで開発を担当するマトリクス型の組織構成のためプロジェクト内での技術的知見があってもそれを他のプロジェクトへ展開していく仕組みが弱く、Android アプリ開発チーム全体としてその知見を活用しての技術課題解決に取り組みづらい状況になっていました。

Column.　退職と人事異動

メンバーの構成変更が日常的かという疑問をもつ読者もいるかもしれません。人的リソースは開発の基礎となる資源です。そのためプロジェクトのキックオフなどで固まってしまえば多くの場合、積極的に変わらない固定要素にみえます。

しかし筆者は技術的課題の考慮外となるとは考えていません。平均勤続年数が２年と仮定した場合、アプリ開発担当者が２４人を超えた時には毎月発生する出来事となるからです。会社が取れるアプローチは開発者が増えるにしたがって平均勤続年数を伸ばすことですがプロジェクトで扱うにはスコープが大きすぎます。現実的にはメンバーの入れ替わりを考慮したアーキテクチャであったり開発体制を整えるべきなのです。メンバーの変更は実際には不確定な事象ですが大規模開発では無視できないファクターであることは理解いただけるでしょう。

7.1.3　技術課題を発見し、チームで話し合う

大規模開発ではさまざまな技術課題がありそうだとわかりましたが、筆者はそれらの技術解決をするためにチームが抱える技術課題についてメンバー全員の目線を合わせることにしました。

開発の初期段階で、複雑なアーキテクチャの理解を助けるドキュメントの整備や新規メンバーのためのオンボーディングフローの策定、実装を手元で動かしてアーキテクチャのあり方を確認できるサンプルアプリケーションの作成ができており、チームに参加してから素早く実際の開発案件に取り組めるようになっていました。

日々の開発を進めていく中で技術的に扱いづらさや不便さについて意見が出るようになったものの、それぞれのメンバーが得た技術的知見を共有して技術課題の改善につなげる仕組みが弱かったため、次のような取り組みを実施しました。

1. チームで扱っている各ドメインから、日々の開発で不便さや扱いづらさを感じる部分についてアンケートをする
2. 静的解析ツールを用いてコードベースに含まれる課題を自動検出し、課題をレビューして定期的に改善案を話し合う
3. チームでランチを囲みつつ外部の勉強会の録画を見たり、チーム内 LT を実施したりする技術的知見共有の機会を作る

はじめに、チームメンバーが実務上どの技術領域で課題を感じ、困っているかを把握するため、次のリスト 7.1 にあげるような簡単なアンケートフォームを用意し技術課題の洗い出しをすることにしました。

このフォームでは、チームがもっている課題感の傾向をつかみ、もっとも課題がありそうな部分を優先順位付けできるよう各領域について課題感の強さを軸に尺度をつくり回答を選べるよう設定しました。またそれぞれの課題について具体的な内容を記述するための自由記述欄も設けています。

●リスト 7.1　技術課題の把握に用いたアンケートフォームの設問例

```
1. 日々の開発のなかで、次にあげる領域ごとに改善に取り組みたいと思う度合いを選択して下さい
  - Reduxを用いたViewModel:
    1. 今すぐ改善に取り組みたい
    2. 改善したいと思っている
    3. 現状で問題ない
    4. 特に気にしていない
    5. どれでもない
  - 画面の振る舞いを実装するController:
    1. 今すぐ改善に取り組みたい
    2. 改善したいと思っている
    3. 現状で問題ない
    4. 特に気にしていない
    5. どれでもない
  - 画面遷移:
    1. 今すぐ改善に取り組みたい
```

第 8 章

素早く安定したアプリを作るためのアーキテクチャの改善

横幕 圭真／@KeithYokoma

本章では技術課題の改善にフォーカスし、実装例や手法を交えつつ根底にある技術課題の改善に対する考え方を伝えます。筆者のチームでの経験から 2 つの事例を取り上げます。

ひとつはユニットテストによる検証不足を解消するためアーキテクチャを改善し、テストケースを充足させていく品質向上の活動です。もうひとつは Android のフレームワークにより順応し、最新のテクノロジーを取り込んでいくアーキテクチャを更新する活動です。

第 7 章で解説したとおり、技術課題の改善には必ず目的があります。ユニットテストの拡充は変更の頻度が高い UI に対して不具合混入を抑止します。UI を構築するロジックをユニットテストで検証して変更時の影響範囲を確認しやすくなります。よって不具合の発生に気づきやすくなり、機能をリリースするまでの開発・品質保証のリードタイムを短縮することが目的といえます。

Android のフレームワークに順応していくためのアーキテクチャの改善は、各機能の設計のばらつきへの対処です。度重なる機能追加や変更によって一貫性が失われてしまう問題に対して、フレームワークというスタンダードな手法に回帰し、一貫性の再導入を試みました。設計の適切な均質化が複雑度を下げることを期待し、最終的に開発期間の短縮を目的として取り組みました。

どのような改善にも目的と目指す理想像があります。本章では筆者が経験した大規模な開発チームでの事例をもとに注目すべきポイントを整理し、素早く安定して開発するという理想を達成する考え方を提示します。わかりやすさのため具体的な事例を通じて技術課題の設定、改善の取り組みを紹介しますが読者が実務で役立てられるように一般化を試みます。

8.1 足りないユニットテストの充足

筆者のチームでは Redux の Reducer と ViewModel のテストを記述していましたが、ユニットテストで検証すべきロジックは Reducer や ViewModel 以外にも存在していました。

Redux の考え方を用いた MVVM アーキテクチャを採用した当初の目的は UI の動きを検証できる設計の実現にあります。画面で取り扱う状態をひとつの状態オブジェクトで表現しておくという原則は、UI を構築するプレゼンテーションレイヤーの実装を「状態オブジェクトから View の設定への変換」に限定できるメリットがあります。原理上、ユニットテストで状態オブジェクトを検証できていれば UI の振る舞いも 100% テストとして表現可能だという考え方です。

しかし実際には、次の問題を抱えていました。順を追って説明します。

1. 振り返るとユニットテストしづらい Controller に UI に関するコードがすべて記述されていた
2. これは状態オブジェクトの表現力が足りないため、Controller にある UI に関するコードが複雑化していたとわかった

Controller は大雑把に Activity や Fragment と似た役割を担っています。いわゆる **God class**（ゴッドクラス）、Android ではおなじみの Fat Activity に類似した問題が起きていることを示していました。

ViewModel 以外の場所で UI の構築ロジックを実装していましたために起きた問題です。本来の設計ではこの ViewModel がユニットテスト対象だと考えており、テストしやすい構造を採用していましたが Controller に対しては考慮が漏れていたということになります。

アーキテクチャ特有の概念である Controller は Activity や Fragment と異なりプラットフォームのテスト支援を受けられません。モックライブラリ[注1)]を使って Controller の動きをモックする場合でもテストの充実を阻害する要素がありました。部分モックというトリッキーな手法を使って開発者が実装した以外の Controller の振る舞いをお膳立てしなければならず、Controller のユニットテストそのものが複雑かつ理解しにくいものになりがちだったのです。

この問題の原因を探ったところ、UI を構築するロジックが肥大化する原因が状態オブジェクトの表現力不足にあるとわかりました。筆者のチームで開発していた機能は新規リリースしたばかりであり、その時点での UX が正しいかの解を持ち合わせていませんでした。

これから最適な UX を探す段階であったため UI を変更する頻度が高くなってしまい、場合によっては AB テストで UI を切り替えて、できるだけたくさんのデータを集めながら検証する必要すらありました。このような試行錯誤を通じた UI ロジックの肥大化は当初の想定から徐々に外れていったのです。

ついには Redux の考え方を用いた MVVM アーキテクチャのメリットである「状態オブジェクトから View の設定への変換」に限定できるメリットが失われていました。

エンジニアが求めている環境は既存の UI の振る舞いに影響を与えない前提を維持し、影響範囲を抑えた複数パターンの UI を作ったり、画面の一部の挙動だけを変更したりと頻繁に変更を加えても他の UI を壊していないことを保証したいという状況です。我々には UI を構築するロジックを論理的な観点で検証できるユニットテストが必要でした。

これらの背景から状態オブジェクトの作り方や構造を改善することがロジックの簡易化に繋がると考えました。最終的な状態オブジェクトの値だけでなく、状態オブジェクトの構造に着目したテストケースを作成できれば、新しい観点でのユニットテスト拡充に繋がります。

この節では前述のような問題を抱えた検証されていない UI の振る舞いに対し、ユニットテスト可能な状態へどのように改善したかを解説します。

注1）　MockK や Mockito などのこと

Column. 部分モックの難しさとリファクタリングの機運

部分モックとは、オブジェクト全体をモックするのではなく、オブジェクトの一部の振る舞いだけをモックすることです。たとえば次のリスト 8.1 に示す PartialMockTarget クラスの部分モックを作成してみると、同じオブジェクトで動作がモックに置き換えられたものと、クラス定義どおりのままのものとが混在した状態になります。

●リスト 8.1　部分モックしたいクラスの定義と部分モックの作成

```
class PartialMockTarget {
  // ユニットテストで返り値を確認したい関数
  fun functionForUnitTest(): String = if (functionNotUnitTest()) {
    return "foo"
  } else {
    return "bar"
  }

  // 部分モックする関数
  fun functionNotUnitTest(): Boolean {
    ...
  }
}

val partialMock: PartialMockTarget = spyk(PartialMockTarget()) {
  every { functionNotUnitTest() } returns true
}
partialMock.functionForUnitTest().shouldBe("foo")
```

複雑で大きくなったクラスの一部の関数をユニットテストする手段として便利である反面、本来ユニットテストで確認すべき関数までモックしてしまいユニットテストの意味がなくなったコードを生み出す危険があります。筆者もこのような状況を何度かみかけました。

Activity や Fragment、Controller を部分モックする場合、自分で実装した部分以外すべてをモックしなければならず、どうしても部分モックの定義に費やすコード量が多くなり、ユニットテストの事前準備が複雑化しがちです。

リスト 8.1 の場合、functionNotUnitTest 関数を別クラスに切り出して委譲すれば部分モックを使う必要がなくなります。多くの場合、部分モックを使う理由はひとつのクラスが責務を持ちすぎているからです。部分モックを使いたくなったら設計の改善を考えるよい機会と捉え、別の責務としてクラスを分離しましょう。リファクタリングに取り組むことをお勧めします。

8.1.1　検証されていない UI の振る舞い

筆者のチームでは MVVM 構成に Redux の仕組みを組み合わせ、サービスが求めるビジネスロジックのテストを記述しやすい構成を目指していました。

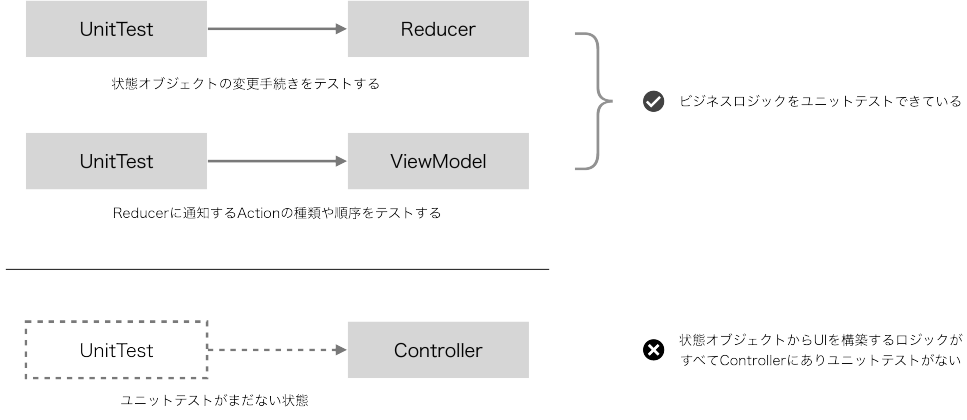
ViewModel は Redux の仕組みを内包し、UI で発生するイベントなどを元に Action を生成して Store に通知し、Store が Reducer で定義した状態オブジェクトの変更手続きを実行します。

状態オブジェクトはそのまま UI の状態を表し、画面に表示するコンテンツを読み込んでいる途

中、読み込み完了、読み込み失敗などの状態を表します。状態オブジェクトの状態を変更するロジックは Reducer が監督しており、副作用のない関数として定義します。

当初は ViewModel や Reducer で作った状態オブジェクトの値を、そのまま UI に表示する想定でしたが実際には状態オブジェクトがもつ値に応じて UI の表示を切り替えたり、UI での利用に最適な値に変換したりするロジックを記述していました。

多くの場合このようなロジックは Controller に直接記述していたため、ユニットテストを阻害していました（図 8.1）。



●図 8.1　テスト対象の整理：Controller のテスト不足が検証を難しくしていた

この技術課題を発見した時にはプロダクトのライフサイクルは新規開発フェーズをとおり過ぎ、グロースのための改善もターゲットでした。筆者のチームでの開発は新機能の追加と改善を両立しなければならないフェーズです。

新機能の追加にあたって既存の UI にも手を加える頻度も高くなっていました。そのため UI のロジックの変更が他のロジックに影響を与えていないかを確認する手段として、プレゼンテーションレイヤーに対するユニットテストが求められており、ユニットテストを記述しやすいアーキテクチャの実現が急務となっていました。

8.1.2　改善の方針

改善の話題に入る前に技術課題をまとめておきましょう。

UI を構築するロジックのユニットテスト不足が課題となって機能開発が遅れるリスクがありました。そこで品質を担保する仕組みが求められています。第 7 章で解説した技術課題に対する取り組みにおける、不足しているユニットテストを拡充していく取り組みにあたります。

課題を作り出している主な要因は UI をテストしづらいアーキテクチャになってしまったことで

あり、Controller に UI に影響するコードが記述されている God class を生み出したことがテスト容易性を低下させています。ここで Controller から UI を構築するロジックを別のクラスへ委譲する設計を取り入れユニットテストしやすくする方法を実践してもよいのですが、もう少し問題を根本から改善します。

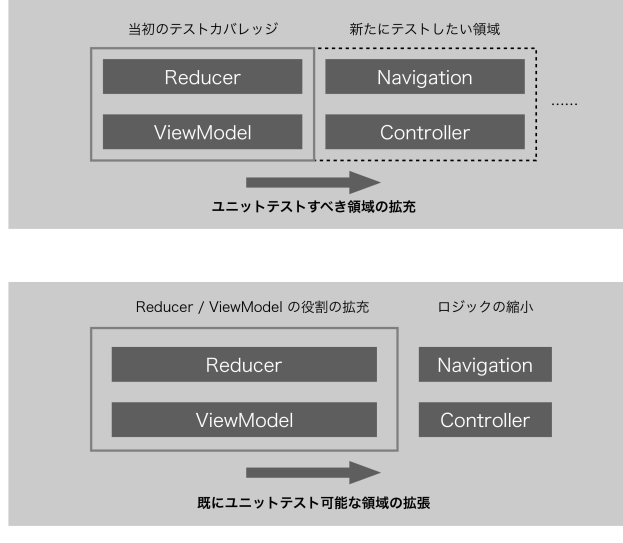
状態オブジェクトをそのまま使って UI を構築するという当初の想定どおりにいかず Controller に直接ロジックが入り込んでしまった理由を探ってみます。

Controller を使いすぎてしまった要因を観察してみると、UI の状態を表現している状態オブジェクトの実装がシンプルすぎたことに起因していました。それを補う工夫が Controller でなされています。

- Controller のなかで状態オブジェクトの値を組み合わせて使う
- 状態オブジェクトの値を別の使いやすい形に加工して UI に反映する

このような処理があっては状態オブジェクトの値をそのまま UI の操作に使える構造になりません。これら 2 点の要因から、不足しているユニットテストのカバー範囲を広げ UI の振る舞いをテスト可能にする改善の方針として次の 2 つを定義しました（図 8.2）。

1. God class 状態の Controller をリファクタリングし、UI の振る舞いを剥がしてユニットテストしやすくすること
2. 状態オブジェクトの持つ値をそのまま UI に適用できる形へ改善し、Controller からロジックを排除すること



●図 8.2　ユニットテスト改善の方針（上図：テスト領域の拡大、下図：Controller 等からのロジック排除）

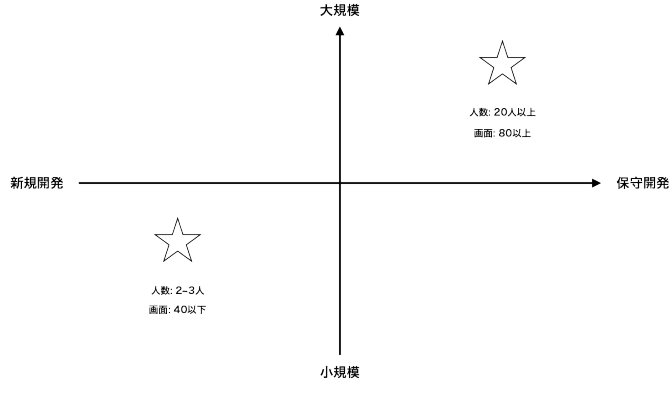
第 9 章

新規開発と大規模開発で何が違うのか

本書ではアプリケーションの新規開発と大規模開発の視点でアーキテクチャに触れてきました。最終章はゼロからスタートする新規開発と運用の歴史を経ている大規模開発という 2 つのフェーズの差について理解を深めます。開発に貢献できるエンジニアの振る舞いについて視点の違いからヒントを得ます。

9.1 アーキテクチャ

新規開発と大規模開発が両立するプロジェクトもありますので改めて関係を整理をします（図 9.1）。アーキテクチャを議論する際に新規開発とした場合の左下の領域を指し、大規模開発といった場合は右上の領域を指すこととします。一般的には新規と保守が対局にあり、開発規模の大小とあわせて 4 領域で類型を作れます。



●図 9.1　本章での新規開発と大規模開発

本来であれば領域ごとに戦略が異なる場合もありますが、新規事業は小規模に展開し、需要の増大やサービスの進展に応じて順次規模を拡大していくアプローチが多く、大規模な事業はその後の保守や運用といったフェーズがサービスのライフサイクルのほとんどを締めています。そのような背景からアーキテクチャの議論でも左下と右上の領域に注目します。

9.1.1　制約の粒度が細かい大規模開発のアーキテクチャ

結論として新規でも大規模開発でもアーキテクチャの役割は**変わりません**。ただし、アーキテクチャがもつ設計上のルールや制約に求められる粒度に違いが出てきます。

大規模開発ではコード量が多く、エンジニアの数も増えてきます。多種多様な考えやバックグラウンドをもったエンジニアが開発に参加すると、それまで暗黙的に保たれていた常識に変化が生まれます。コードスタイルやドキュメントの書き方、レビュー手法など細かなところにも違いが表れる可能性があります。その中で秩序のあるコードを保つには粒度の細かなルールが必要になってきます。

やっかいなことにルールを細かく定めただけではアーキテクチャは機能しません。ルールが正しく運用されているかというチェックに人的リソースが必要になり、その分のコストも増加します。そういった人的コストを減らし、能動的にルールを守るだけではなく強制的に守る仕組みも必要です。

たとえば Android アプリのプロジェクトではモジュールを分割することで依存関係を強制させるという工夫もそのひとつです。大規模開発では同時並行に多数のエンジニアが作業することから、新規開発と比べるとやり過ぎに感じるくらいの数のモジュール分割が進みます。

新規開発では大規模開発と同様の細かさをもって制約を作ると不要なルールを多く設けてしまうことになります。チームメンバーにとって大きな問題ではない行動をルール化して運用をチェックしても、それらは単純に時間の無駄となってしまいます。

また分ける動機がないにもかかわらず、細かくモジュール分割しても実装時の無駄やコード量が増えてしまいます。新規開発で大規模開発とで比較することにあまり意味はなく、同等のものさしで測れるものでもありません。大規模開発事例を参考に新規開発のアーキテクチャを選択する、またはその逆を行う、どちらのケースも開発チームにマッチしない結果を迎えます。

9.1.2　成長痛を軽減する新規開発のアーキテクチャ

新規開発と大規模開発はまったくの別物といえるでしょうか。サービスには新規開発のタイミングがあり、成長を重ねて大規模開発フェーズに到達していくことを考えると地続きといえるでしょう。どんなプロジェクトも大規模開発にたどり着くというわけではありませんがサービスが拡大に至るまでには少なからず、成長痛があります。ここでの成長痛とはコードがカオス化してしまってリアーキテクチャやリファクタリングをしなければならない状況に陥ることを指しています。

サービスの拡大に備えるためにも新規開発のアーキテクチャは適当なモノを選んではいけません。世の中には新規開発ではそれなりのアーキテクチャを選んで２、３年後にフルスクラッチで作り直せばよいという意見があることも承知していますが筆者は経験上、反対の立場です。

それなりの度合いにもよりますが適切な制限が用意されていないアーキテクチャは２、３年も持つことなくカオスなコードを生み出す場合がほとんどです。

有望なサービスであればあるほどリリースから２、３年後は成長フェーズにいることが多く、そ

のようなタイミングで新機能の追加や施策をストップしてフルスクラッチする期間を設けなければならないというのは事業にとって大きなマイナスとなります。そのため、大抵のサービスではフルスクラッチという手段をとれないことが多いです。

結果として計画していたアーキテクチャ書き換えのタイミングを逸し、さらにカオスになっていきリリース当初にいたエンジニアは辞めてしまい、また採用もできず、たくさんの施策を打って成長フェーズにギアを上げたくても開発スピードが上がるどころかむしろ下がっていってしまうといった地獄のような展開がやってくる可能性があるのです。

新規開発フェーズからサービスが成長を重ねて、大規模開発フェーズに至るまでには少なからず、どのステップにも成長痛があります。フルスクラッチという非連続なジャンプを求める気持ちは筆者にも理解できますが成長痛への注意を怠ってよい理由にはなりません。新規開発で、もっというならば最初のリリース時点でアーキテクチャがどれだけチームに生産性をもたらしているかによって今後の成長痛の強弱が変わってきます。

筆者は４、５人で新規開発したサービスが急激に成長し数年以内に２０人以上の大規模開発に至ったプロジェクトの経験があります。そこでは新規開発時にアーキテクチャを適切に選んでいてよかったと身に染みる局面が何度もありましたし、準備をしていたにもかかわらず多少なりとも成長痛を体験しました。

新規開発、大規模開発それぞれ求められるアーキテクチャの粒度、細部が異なるという主張は前項のとおりです。大規模開発ではチームメンバーのバックグラウンドに多様性が生まれることからも細かいところまで制限を加えて、知識を均質化することがチームの生産性の最大化に寄与します。これは大規模開発で生産性を低下させないわけではなく、生産性の低下を**最小限に抑えて**歩み続けるためにアーキテクチャを使うということです。そのまま新規開発に当てはめると生産性を落としてしまいます。だからといって新規開発でのアーキテクチャをないがしろにしてはいけません。新規開発のアーキテクチャ選定がサービスの成長を支え、大規模開発に至るかもしれない可能性の年月のなかで感じる成長痛の度合いを決めるのです。

9.2 チーム

新規開発と大規模開発においてチーム編成、人にかかわる点についてどういう差があるのかもみていきましょう。

9.2.1 組織編成について

新規開発では表 9.1 のようなチーム編成をとることが多いです。ひとつのチームにさまざまな職能領域の開発者が集まるクロスファンクショナルチームです。

●表 9.1　新規開発でよくあるチーム構成

領域	人数
Android	1〜2人
iOS	1〜2人
サーバーサイド兼インフラ	1〜2人

他方で大規模開発では2とおりのチーム構成をよく目にします。まずは職能領域ごとにチームを構成するパターンです（表 9.2）。職能別チームやコンポーネントチームと呼ばれるパターンです。

●表 9.2　職能領域ごとの構成

チーム	領域	人数
Android チーム	Android	8人~
iOS チーム	iOS	8人~
サーバーサイドチーム	サーバーサイド	10人~
インフラチーム	インフラ	3人~

次に紹介する表 9.3 が開発する機能ごとにチームを構成するパターンです（表 9.3）。機能別チームやフィーチャーチームなどと呼ばれ、第 5 章で紹介したチーム編成はこちらのパターンに近い組織構成をしています。新規開発で見られるクロスファンクショナルチームが同時に多数存在するパターンです。

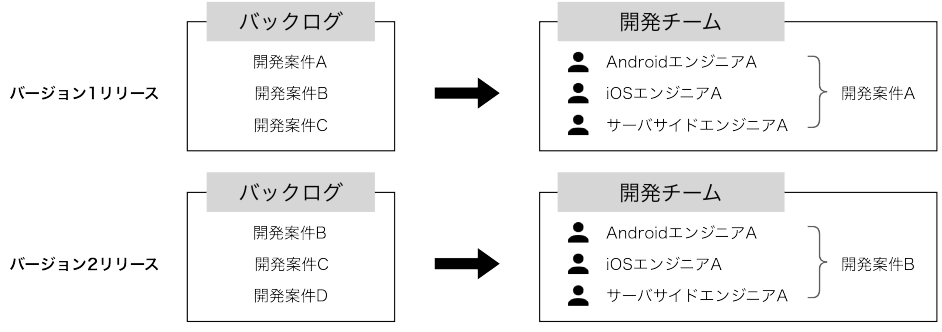
●表 9.3　大規模開発でよくある機能別の構成

チーム	領域	人数
機能 A チーム	Android	1〜2人
機能 A チーム	iOS	1〜2人
機能 A チーム	サーバーサイド	3人
機能 B チーム	Android	1〜2人
機能 B チーム	iOS	1〜2人
機能 B チーム	サーバーサイド	3人
機能 C チーム	Android	1〜2人
機能 C チーム	iOS	1〜2人
機能 C チーム	サーバーサイド	3人
インフラチーム	インフラ	4人

どれかがアンチパターンであるとは一概にはいえません。会社や組織構成によってどういうチーム編成がよいかも変わってくるでしょう。これらのチーム編成のパターンごとに得手不得手があることは覚えておくべきです。

新規開発においては開発者全体の人数が少ないため、職能や機能による分割はせずひとつのクロスファンクショナルチームとして組成します。プロダクトの継続的なリリースに比重をおいて開発

を進めます（図 9.2）。

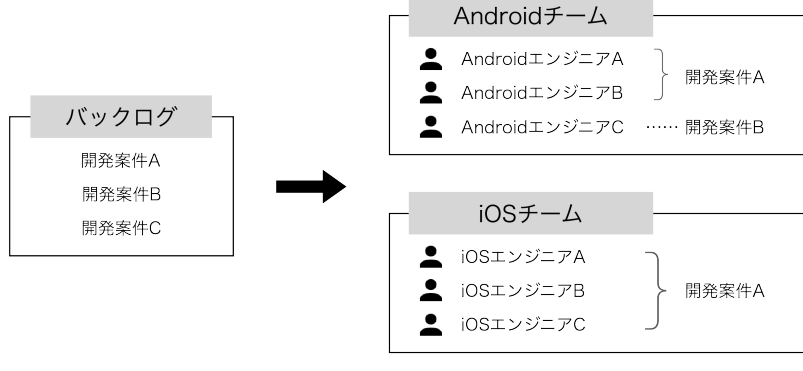


●図 9.2　新規開発チームと開発バックログによる案件管理

チームの人数が少ないということは開発に投下できる工数が限られていることを示します。価値の高いことから順番に取り組む必要性が生まれるため、プロジェクトのタスクは優先順位をつけて管理しているでしょう。

実装の順序など開発ワークフローもシンプルになります。これは同時にいくつもの施策を並行することには向いていません。効果のある取り組みを求めてフットワークが軽く、コミュニケーションを密に取る組織になるでしょう。

会社組織自体がある程度成熟した状態やサービスが成長して職能領域でチームを編成する場合、Android、iOS、サーバーサイドなどプラットフォームごとに開発案件や技術課題を管理します（図 9.3）。



●図 9.3　職能別チームと開発バックログによる案件管理

著者

釘宮 愼之介 @kgmyshin

新規開発でテックリードとして携わることが多いAndroidエンジニアです。どうすれば良いスタートダッシュが切れて、チーム全体の生産性を恒常的にあげることができるのか考えてきました。読後に「これは今のうちのチームにすぐ取り入れたほうがよさそうだ」と思えるような事柄が少なからずある内容になるように、再現性のあるノウハウを惜しみなく、本書に詰め込みました。

横幕 圭真 @KeithYokoma

スタートアップでAndroidアプリをつくったり、複数のチームにまたがってひとつのAndroidアプリをつくったりしています。気がつけば10年にわたってAndroidアプリ開発を続けていますが、振り返るとプロダクト開発も技術的挑戦、課題解決もそれぞれに楽しんで取り組んできました。本書では、大きな組織でプロダクト開発と技術的挑戦、課題解決の両輪を同時にすすめ、特にアーキテクチャという面でチームと技術を前進させるノウハウを盛り込んでいます。

編集

日高 正博 @mhidaka

DroidKaigiの代表理事、技術書イベントの技術書典の主宰。技術の共有やコミュニケーションに興味があり、TechBoosterのブログや書籍、講演活動を通じて開発情報を発信してます。アプリ開発の困難をチームで乗り越えるエンジニアのための一冊を届けたいと編集者として関わりました。