CureApp Advent Calendar 2023 1日目

@shinout (しん すずき) in 株式会社CureApp

# 9年開発を牽引して見えてきた、共通化すべきものと個別でつくるもの

設計 経営 組織 DRY 共通化

最終更新日 2024年03月26日 投稿日 2023年12月02日

## はじめに　〜DRY or WET〜

怠惰であることが美徳とされるエンジニアは、一度つくった機能は二度と作らないようにしたいものだ。
だから共通部分を見つけ出し、それを切り出してライブラリとして使い回すということが行われている。

- コピペはするな。
- DRY(Don't Repeat Yourself)

ところがある程度の経験を積むと、

- むしろコピペしろ。
- WET(Write Everything Twice)

といった状況もあることに気づく。

この両極の価値観は、少なくとも「何事も共通化すればいい、というわけではない」ことを教えてくれる。
問題はその使い分けだ。

これに関しては実に多様な意見がある。
本記事ではそれらを類型立てて紹介することはせず、かわりに私の経験に基づく考えを

私は9年間、同一の組織で開発を牽引する立場におり、共通化と個別化の判断を繰り返し、またその結末も見てきた。それらの事例と、そこからの学びを紹介したい。
いくつかはすでに言い古されたものと思うが、もしかしたら読者にとってはじめての観点もあるかもしれない。皆さんの日々の開発に気づきを与えられれば幸いである。

## 私のいる組織について

私の学びは、私の経験した事業環境に大いに依存している。
だから別な事業環境を生きる読者が私の知見を導入するには、適切な翻訳が必要になる。
そこで、まずは私の組織と事業環境について紹介したい。その差異を理解しながら、自身の立場に置き換えて考えてみてほしい。
面倒な方は読み飛ばしても構わないが、私の提示する学びが信じがたい場合は、事業環境の差異について振り返ってほしい。

### ヘルスケアではない「治療アプリ」の会社

私は株式会社CureAppという会社を共同創業した。私の役割は開発を統括する立場である。
2014年に2人で始めた医療系のスタートアップで、2023年12月現在では200名ほどの組織になっている。そのうちソフトウェアエンジニアは現在40名ほどだ。

私達は医師が患者に処方する、病気を治すためのスマートフォンアプリ「**治療アプリ**」を医療機器として開発し、販売している。
例えば禁煙治療や高血圧治療のためのアプリだ。
治療アプリはヘルスケアアプリと以下のことが違う。

- 開発開始からユーザーが利用するまでに**5年以上**かかる（治験や薬事承認などの壁があるため)
- ユーザーが利用してからは、**大きな改善は加えられない**(承認を受けた版を提供するため)
- 治験で治療効果を示し、**国に承認を得る**ことが必要
- 医師用の管理画面があり、医師が患者に処方する**医療機器**である

### 複数プロダクトを並列で開発する環境

上記のような事業領域であるがゆえに、私たちは複数の疾患に向けて個別のアプリ開発を並行して行なってきた。
つまり、**複数の独立した事業およびプロダクトが並行して作られる環境**にあるということだ。
このような状況のなか、共通化すべきものと個別でつくるものという検討が頻繁になされてきた。以下にどのような共通化、個別化判断をして、結果どうなったか、の具体的な事例を示していく。
★の数は、共通化してよかったものほど多い。

## 1. クロスプラットフォーム開発 ★★★☆☆

起業時点でアプリ開発未経験(!?)だった私が選んだ戦略は、単一ソースでiOSにもAndroidにも対応することだった。学習コストの軽減と、開発工数の大幅な削減が狙いである。トレードオフにしたものは、ソフトウェアの安定性や、OSの最新機能を用いることである。

### React Nativeで確実な共通部分を増やす

2015年頃にReact Nativeが登場し、クロスプラットフォームの価値は圧倒的に向上した。
FluxアーキテクチャはUIが論理的に表現されているため、OSによらない領域が増えたのだ。
つまり、OSによらない論理演算の世界にアプリの大半を持っていくことができれば、OS間でのコード共通化は実現がより容易になっていく。
嬉しい副作用として、論理演算に閉じる世界とそうでない世界の境界を意識して開発するすることができるようになった。これはドメイン駆動設計と相性がよかった。ドメインのコードは当然論理に閉じており、外部依存の部分は何らかの方法で外部から依存性を注入すればよいという発想だ。

### ローカルfirstなアーキテクチャが実現可能に

論理演算に閉じるコードを、クライアントのOSによらず共通化できるとすれば、クライアントにビジネスロジックの大半を持ったアーキテクチャが実現できる。これは私

行われるため、それはローカルで完結できる。するとオフラインでも利用可能なアーキテクチャであるとか、サーバーとの状態管理の困難さ解消や、操作環境の体験向上、安定性に寄与した。

## 突き詰めた設計原理は入れ替わる組織には向かない

ただしこれらの高度かつあまり馴染みのない設計原理の真意をチームが理解するのは難しく、一度そのルールのviolationが起こると綺麗に保っていた意味がなくなり、ただの厄介な設計になり、あとから加わったエンジニアの非難の対象となることもあった。
これはクロスプラットフォーム選択によるものというよりも、クライアントコードの共通化の設計原理を過度に突き詰めすぎたことに由来すると考えている。

## フレームワーク依存の難しさ

クロスプラットフォームは、それを実現するフレームワークに依存し続けてしまうという窮屈さはある。
特にフレームワークのバージョンを上げるための労力は一定支払い続けなければならない。
とはいえ、それはネイティブでも一定必要なメンテナンスコストでもある。
自身の事業環境からベストな選択を検討し続けることが必要だ。
事業フェイズが変われば、選択するべき技術も異なるのは当然である。
**技術選定は、好みでなく、経営判断**であるということだ。

## 2. UIパーツの共通化 ★★☆☆☆

React Nativeでちょっと複雑なUIパーツを、1ファイルで外部依存なく完結できたとき、
それはなんとも言えぬ満足感を与えてくれる。
モジュール性。非依存性。疎結合性。
プログラマーにとってこれだけ極上の玩具はない。

私たちは社内npmパッケージにこれらのパーツをカタログ化した。

しかし、この取り組みは大きな生産性の向上には寄与しなかった。

## UIはUXのためのもの

仮に移植性の高いUI部品を作れたとして、他のプロジェクトはそれを自然に受け入れるのか。
答えはNOだった。

例えば体重記録のためのクールな目盛り式UI。
体重記録は、どのようなプロダクトであっても必要な機能だ。
にも関わらず、見た目も含めてそのまま受け入れることはできないのである。

なぜならプロダクトオーナーは、アプリの継続利用、そして治療効果のために、与えたいUX(ユーザー体験)をまず考える。デザイナーはその実現のためにUIを考える。
**「ユーザーから始めよ」と「あるんだから使おうぜ」のどちらを優先するのか。**
これも経営判断だ。
あとから仕様の変更がしにくい性質をもつ治療アプリ。
そして治療効果が出るかどうかが治験の一発勝負である治療アプリ。
このような事業環境だと、「あるんだから使おうぜ」は退かざるを得なかったのである。

## カスタマイズするぐらいならコピペしろ

Reactコンポーネントは、パラメータを外部から注入することで、その見た目のpreferenceを設定するようにも設計できる。
UIパーツの色や形、文字サイズなどをカスタマイズできるようにすれば、異なるプロダクト間でも見た目をそれぞれに合わせることができる。

しかし、そのような努力をするなら、コピペしたらよい。
**コピペは、ノウハウの忠実な再現**だ。
そして、**コピペは結合を切ってくれる。**

結合があることが開発を加速させるのか、はたまた足枷になるのかは、状況次第だ。
UIパーツの場合、共通で修正しなければならない部分よりも、個別で改善していく部分のほうが多い。

## 3. Atomic Design ★☆☆☆☆

つい5年ほど前にはAtomic Designという考えが流行した。
これは、原子-分子-生体といったもののアナロジーで、UI部品を細分化して統一感をもった開発をしようというものだ。

これも実際にはコピペでよかった。
UIの場合、結合がないことが重要だった。
原子に相当するような普遍的なUI部品を修正すると、影響範囲が無限大だ。
それらすべての見た目がうまくいっているのかに、修正者は責任を持てるのか。
逆に、どうしても修正すべき内容があれば、コード内一括置換でよい。

## 4. UIと分離不能な機能 ★★★☆☆

※この章は、秘匿化すべき技術があるために、奥歯にものが挟まった言い方になっているせいで、面白くない可能性があることに留意

どの治療アプリも、患者に個別化されたメッセージを送る。
その抽象的な機能をライブラリにしようという発想は、あって当然だ。

しかし、これをどのように共通化すればよいのかということには苦心した。
共通化したい部分と個別で作りたい部分は以下のようになる。

### 共通化したい

- 患者の情報を把握する機構
- 状況に応じて適切なメッセージを表示する機構

### 個別で作りたい

- 情報取得、およびメッセージの表示UI
- 情報取得のロジック、また状況に応じたメッセージの表示ロジック

## ロジックは注入できた。UIは？

ロジックの注入を行う機構はうまくできた。その功績は大きく、それがゆえに現在でもワークしている仕組みである。
問題はやはりUIである。
特定のUIを想定した機能であるが、機能が複雑であるがゆえに、UIの制約も大きい。
UIの自由な注入は、論理的には可能にみえるが実際の運用では課題があった。

## 設計思想の継承

このコア部分のライブラリは、別な問題も引き起こした。
この開発を牽引してきたメンバーが退職すると、この重厚なライブラリの影響範囲の大きさゆえ、気軽にメンテナンスするのが怖くなってしまった。

ソフトウェアエンジニアリングは、コンピュータサイエンスではない。人間の性質を考慮するべき領域だ。
このような人間の性質を考慮するならば、共通化すべき部分の設計思想が理解可能な形で継承されるためのしくみを築くべきなのだ。
一方で、属人化が完全に排除できないのがスタートアップだ。
優秀なスターが引っ張ることで開発が非線形の伸びを見せるのもまた、スタートアップの戦い方だ。
**共通化ライブラリをスターに任せることは、スタートアップの劇薬だ。**

## 5. ドメイン知識によらないサーバー-クライアント間通信フレームワーク　★★★★☆

ここまで来ると何のことを言っているのかいよいよ分からない。
そんなふわっとした概念を複数チームで共通化するということ自体がまず困難だ、と思うかもしれない。

しかし、それをやってしまったのが、ReactでありGraphQLだ。
彼らは自社技術を汎用的なものとして、ドキュメントをメリットとともに伝え、
イケてるものであることを主張し続けることで、突飛な概念をエンジニア界隈に根付かせることに成功している。

ソフトウェアエンジニアリングは人間のものだから、人間に理解されるための言語化や努力が不可欠なのだ。

とがあるはずだが、
それを説明して理解させるのは本当に難しい。

## で、結局これはなんなのか

私たちが取り組んだのが、ローカルFirstなアプリにおける通信技術の共通化だ。

### ローカルFirst？

ローカルFirstとは、ローカルのデータがまず更新され、その後サーバーにデータが保存されるというタイプのアーキテクチャを指す。私の過去の記事
https://qiita.com/shinout/items/5ff2ecf7a49258eee022
などにも詳しい。

これはまるで一人でコードをGit開発するようなものだ。
常に自分のコードが最新だが、それでもGitHubにpushする目的は、自分のPCが壊れても大丈夫なように、そしてコードを誰かに見てほしいからだ。

同じように治療アプリは、患者が自分の利用記録や行動記録をアプリに入力する。
その入力内容は常に手元が正しいが、ログアウトしてもデータが失われないように、
また医師にデータを見てほしいという理由でサーバーに保存するのである。

## フレームワーク開発と浸透

そのような大きな設計思想を実現するべく、私はフレームワークを開発した。
https://github.com/phenyl/phenyl

phenylはオープンソースで利用できる、ローカルFirstをGitのようにコミット差分の通信で実現するためのフレームワークだ。これによって、あらゆるデータをサーバーに保存するためのAPIを 1 つひとつ作らなくて済む。

1年間ぐらいかけて、業務時間外だけで作った。
自分の30歳ぐらいの余暇を捧げた、私にとっての青春である。

とはいえ思い入れは品質や実用性とは無関係であるし、上述のとおり大きな問題は概念の浸透であった。

だが、賢いメンバーたちはそれが何であるかを僕の言葉の節々から把握し、放っておいたらこれを使ったシステムが完成していた。

たまたま優秀なメンバーに助けられて浸透したが、やはり人の入れ替わりとともに、この存在意義の浸透は課題であり続けてしまうだろう。

## 創業者フレームワークが経営判断を狂わせる

phenylは実際、OSSの皮を被った創業者フレームワークだ。
私は独裁的な組織運営をしているわけではないが、これを真っ向から否定するのにもエネルギーが要る。
それは新しいイノベーションにならない。

だから私は同様のニーズが満たせるApollo Clientなどによるアーキテクチャを新規プロダクトではむしろ選んでもらったりしていた。しかし、これは組織にとって本当によい選択だったのか、バイアスだらけの当事者の私には判断が難しかった。

つまり創業者による共通化コードは、良くも悪くも技術選定つまりは経営判断を狂わせるだけのパワーがある。
そこから脱却するには、このコードのオーナーシップを委譲するほかない。
その作業を怠ったまま、開発現場から離れた偉いだけの人物になってはならないぞ、という自戒の念を今込めた。

# 6. 言語 ★★★★★

ここからはライブラリの話ではない。技術選定の共通化・個別化の話になる。

私たちはすべてのプロダクト開発をほぼTypeScriptで完結させている。
明らかにこれは正解だった。

## プロダクト間での言語統一

言語が統一されることで、以下のことができる。

- プロダクト間でのライブラリの共有ができる

- 社内勉強会で濃い話ができる
- 採用のときにターゲットが明確になる

デメリットもある。

- React Nativeなどアプリ側がフレームワーク依存する
- Rustが触りたいのですが？→ごめん。
- Flutterが触りたいのですが？→ごめん。
- 分析はpythonだよね？→だよね。

## pythonも避けて通れない問題

私たちはアプリケーションエンジニアとデータエンジニア、データサイエンティストを、組織ごと分けている。
データエンジニア、サイエンティストはpythonを用いてデータを扱い、分析する。
その話も面白いが別な機会に紹介する。

## Flutterがやりたいときはどうするのか

昨今はクロスプラットフォームのアプリ開発フレームワークとして、Flutterが名声を集めてきている。
技術者としては、面白いものができているな、と感じる。

しかし経営者としては、
「**そんなもの、流行ってくれるな、頼むから**」という相反する感情も懐いてしまう。
React Native/TypeScriptなエコシステムの相対的価値の低下を招くからだ。

しかし、そんな願いに意味はない。
だから環境に応じて判断するだけだ。

意地を張って既存のアーキテクチャを守るのか、パイロット的な導入を認めるのか。

COBOLがガラパゴス化したのは、言語を刷新する経営判断をしてこなかった結果である。
経営者は「動いているからいい」の１段2段上の議論をしなければならない。

その技術は安定した品質を届けられるのか。
その技術は市場の活きの良いエンジニアにとって魅力的であり続けるのか。
新しい技術の寿命は？

これが技術トップが考えるべきことだ。

## サーバーにTypeScript(Node.js)を用いることの是非

癖の強い非同期ネットワーク言語として生まれたNode.jsは、14年経って丸くなった。
async/awaitなどの言語仕様の変更を経て、非同期波動拳コードは姿を消し、だいぶ使いやすくなっている。

処理速度に関しては「推測するな、計測せよ」で考えたい。
Rust/Goといったコンパイル型言語を用いる必要性は運用しながら考えればよいだろう。
サーバーでシビアな速度が要求されるビジネスかどうか、というところは技術選定のポイントだ。

### サーバー・クライアント間での言語統一の目的

サーバー・クライアント間で言語を統一したことで受ける恩恵は、実際に何だったか。

まず、明確に違っていたことでいうと、コードの共通化だった。
型情報は共通化することはあるし、いざとなればどこでも動くコードを保持しておくことに価値はあるが、
無理やり共通化すべきことはなかった。

結局得られた最も価値のあったことは、エンジニアのアイデンティティの形成であった。
「自分はサーバーエンジニアだ」「自分はフロント」といった帰属意識から、
「自分はXXというプロダクトのエンジニアだ」という帰属意識に変化させることだった。
得意不得意はあれど、同じ言語で全領域を担当させることで、プロダクト全体を担う人材を育てることに繋がった。

## 7. インフラ　★★★★★

インフラ層の共通化はやはり価値が高い領域だった。

- コスト管理の容易性
- SREによる横断管理の容易性

ここは私たちの組織としてはまだ論点が少ない部分である。
サーバーの重要性がより高い組織であるとより解像度の高い議論ができていることだろう。

## 8. 開発環境　★★★★★

コード管理、タスク管理、エディタなどの共通化について。

コード管理については、GitHubを用いることに何の疑問も持たずにここまで走ってこれた。
それ以外の選択肢が検討の俎上に上がった記憶がない。

逆に、タスク管理については、大きな変遷があった。
GitHubに連携していたほうがいいのか、していないほうがいいのか、とか。
デザイナーも入れたほうがいいのか、入れないほうがいいのか。
スクラムって何？このあたりは8章の組織体制のところでも少し触れたい。

エディタについては2018年頃からVS Codeが特にTypeScriptにおいては他者を圧倒する利便性を発揮していたことから、こだわりで選ぶだけの時代でもなくなったように思う。

## 9. 組織体制　★★★☆☆

もはやこの章はコードの話ですらない。
だがソフトウェア・エンジニアリングが人間の営みである以上、組織体制もエンジニアリングの対象なのだ。

各プロジェクトごとに組織体制を同じようにつくるのか、それとも中の人間に合わせて柔軟に扱うのか、といったことだ。
この件については下記の記事に詳しく書いているので、ここでは簡単に触れるにとどめたい。
https://cureapp.blogspot.com/2023/10/blog-post_30.html

## EMとTLとPM

開発組織において、リーダーシップをとる役割は多数必要だ。
エンジニア領域に限っても、EM（エンジニアリングマネージャー)とTL(テックリード)とPM(プロジェクトマネージャー)の3要素がある。
EMは人のマネジメント、TLは設計実装のマネジメント、PMはプロジェクトのマネジメント。
これらの役割を、誰が担うべきか。
当たり前に聞こえるかもしれないが、過去に全部1人に任せていたら、負担が大きすぎたこともあり、結果的にエンジニアの大量離職という大事件を招いてしまった。
過去にたまたまうまく行っていた体制を振り返ると、そのときは属人的に能力のあるエンジニアがこの中の2つ3つを勝手に兼任してくれていたのである。
その成功体験をすべてに共通化しようとしたら、大きく転んでしまったのだ。

## 属人性との正しい距離感

会社というものは、属人性を廃し、業務を標準化するものである。
しかし、そのように個を扱うには人間の特徴はあまりにも違いすぎる。

仮にEMもTLもできるメンバーがたまたまいれば、そのメンバーに両方の役割を頼ってもよいのか。

答えは、短期ではOKで、長期ではNG、ということだ。
短期では最大のパフォーマンスのために、仕方なくOKするのだ。
長期では「個人には依存せず、組織として回る」という~~あるかどうか分からない~~理想のために標準化のために動くのだ。
そして、短期と長期の境目は、他でもない経営のさじ加減なのだ。
ずるずると「短期」を続けていれば、そのスーパープレイヤーは転職するかもしれない。
すぐに「長期」を整えると見栄えはいいが、費用対効果やフェイズと合っているのか。

適切な距離感の見極めもまた経営判断である。

## 終わりに

UIなど、ユーザーから考える部分は個別的な対応のほうがよい。
逆に、通信アーキテクチャなど、領域に依存しないものは共通化が検討可能だ。
そしてインフラや開発環境、言語など、選定によって製品の自由度が失われにくい部分は積極的に共通化し、その恩恵を受けるとよいだろう。

## 最適解の模索。それは楽しい。

共通化と個別化のさじ加減が難しいのは、つまるところソフトウェア・エンジニアリングが人間の営みであるということに起因するのであった。
すると例えば生成AIがコードの9割を書く時代になったら、状況は一変する可能性が高い。
経営判断を求められる技術トップは、この最適解を模索していく必要がある。

それは楽しいことだ。なぜなら組織ごとに答えが違うからである。



**@shinout (しん すずき)**

CureAppの中の人。 マルチプラットフォーム、Universal JSのことに関心あり。

フォロー

**株式会社CureApp**

医学的エビデンスに基づいた医療機器プログラム『治療アプリ』を開発しています。

https://cureapp.co.jp/

フォロー

## 今日のトレンド記事

@DEmodoriGatsuO (Gatsuo De'modori)
2024年05月18日

### Power Apps & GPT-4oを使って超高速で画像解析アプリを作る！

PowerApps AzureKeyVault PowerAutomate ChatGPT GPT-4o

♡ 37


@naoya_347 (なおや)
2024年05月18日

### useSession？なにそれおいしいの？

React Next.js useSession

♡ 20 9

@kaku3
2024年05月18日

### AIは仕事ではなく仕事力を奪う？

ポエム　教育　AI　pm　新人プログラマ応援

♡ 15

@mkt_hanada (Makoto Hanada)
2024年05月18日

### 【AI品質・AIテストまとめ】AIシステムの品質を高めるために

テスト　AI　データサイエンス　品質

♡ 17

@sanjushi003
2024年05月17日

### VMware Fusion 環境 (macOS) に Windows Server 2022 仮想マシンを作成してみた

Mac　Broadcom　vmware　VMwareFusion　WindowsServer

♡ 7

トレンド一覧を見る

## 関連記事 Recommended by

**宣言的UIはReact Hooksで完成に至り、現代的設計論が必須の時代になる**
by erukiti

**Webとスマホネイティブの間にある技術（ガワネイティブやハイブリッド、クロスプラットフォームなど）**
by noboru_i

**「一人で書いて、一人で使うコード」しか書いたことがない新卒web系開発エンジニアがだいたい必要な技術...**
by hokkun_dayo

20　9

## コメント

この記事にコメントはありません。

T コメントする

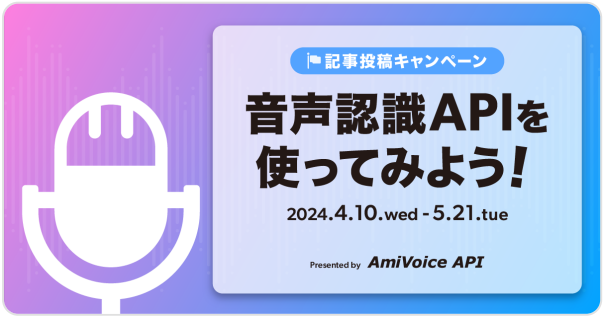
プレビュー

コミュニティガイドラインに基づき、良識ある内容を心がけましょう。

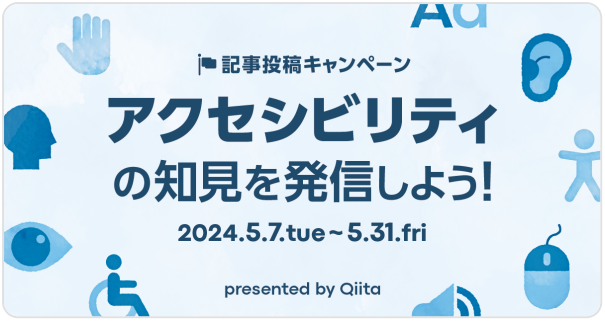
テキストを入力

0B / 100MB 投稿する

## 記事投稿キャンペーン開催中



詳細を見る




**アクセシビリティの知見を発信しよう！**

2024/05/07~2024/05/31

詳細を見る

すべて見る



20 9