設計の話 DDDとMVP

自己紹介

釘宮 愼之介

くぎみや しんのすけ

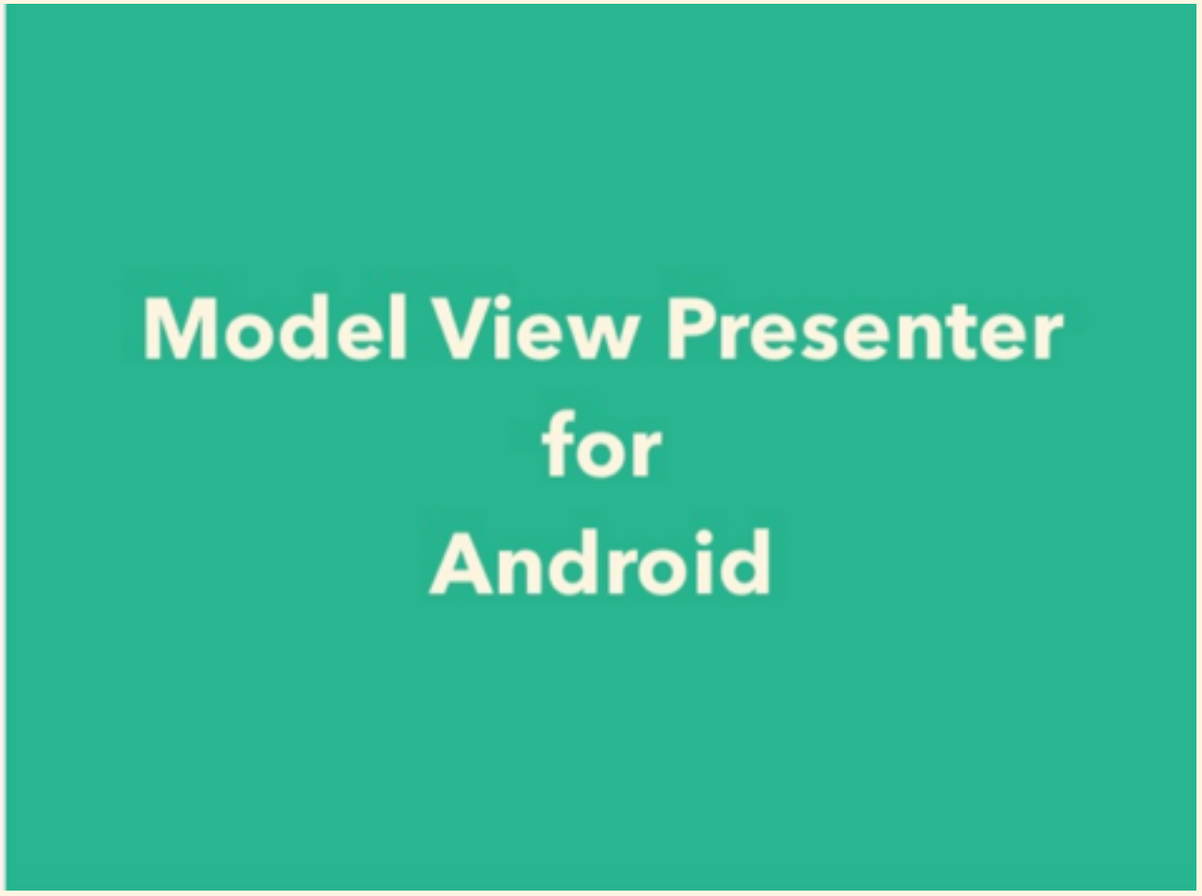
リクルートマーケティングパートナーズ

Androidエンジニア

@kgmyshin / 有象無象


これからの設計の話をしよう


NET BIZ DIV. TECH BLOG
PRODUCED BY Recruit
[Android] - これからの「設計」の話をしよう
釘宮慎之介
2015.06.12
233
モバイル
58
2
83
90


Model View Presenter
for
Android


CQRSの考え方を
モバイルに適用してみる
(試行錯誤中)
potatotips #28

振り返ってみると

設計に関わる発表や記事が多かった

その結果ここに呼ばれたのかなと

思ったので

設計の話をします

agenda

- 設計の話を聞く前に
- プログラミングにおける設計
- 複雑性と設計
- MVPとDDD
- 各レイヤーと各クラスの説明
- まとめ

設計の話を聞く前に

設計の話を聞くと、不安になる

- 自分の設計が否定された気持ちになる
- 今やってる仕事の設計全部間違ってるってことか？
- 全部書き直しか？
- 政治・宗教・野球・設計

ざわ…

ざわ…

安心してください

- これから話す内容は、あくまで一意見で、僕がこういう設計だと良いかもしれないっていう**一つの例**にすぎません
- 聞いている人たちの今の設計を否定するものではありません。
- **アイドルの推しの話してるんだ**くらいノリで聞いてください
- 参考になりそうなところだけ、知識として吸収したり、参考にしてもらえると有り難いです。

この資料は公開はしません

- ネットにあげるとここまで書いても、これらのコンテキストを共有できない人が多く、精神衛生上よくないので…

- 資料欲しい人は、 **@kgmyshin** までメンションいただけると即渡します

プログラミングにおける設計

設計とは

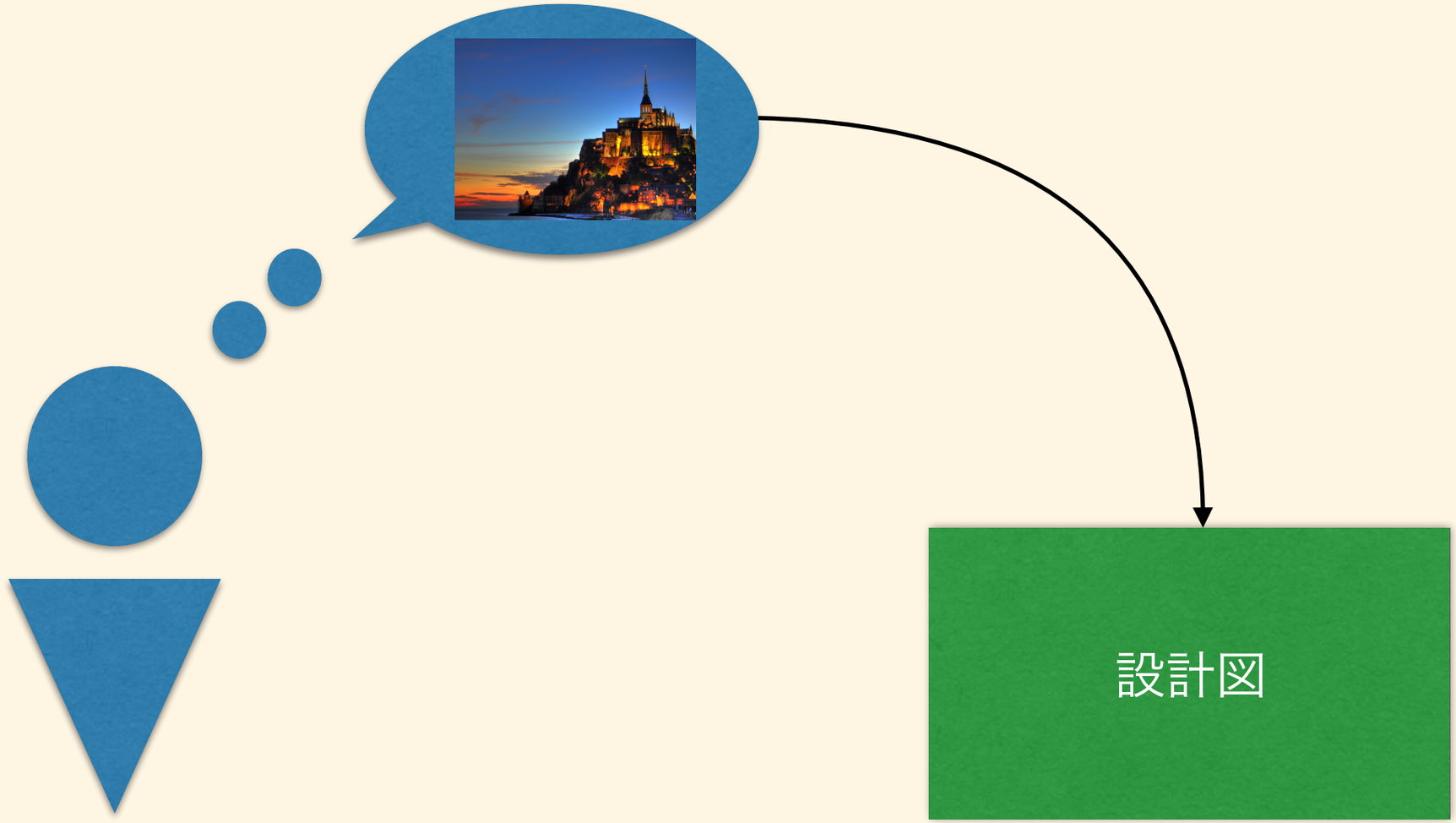
設計とはそもそも

プログラミングに

限ったことではない

一般的な設計の目的とは？

一般的な設計



イメージしたものを実現するまでの見通しを立てる行為

プログラミングにおける設計とは？

プログラミングにおける設計の目的

同様に

イメージしたものを実現するまでの

見通しを立てる行為

一般的な設計と比較した
プログラミングの設計の難しさ

プログラミングにおいては、

完成した後も

ものが変わり続ける

そのため

どうすればイメージ通りのものとなるか？

だけでなく

どうすれば変化に強いか？

も求められる

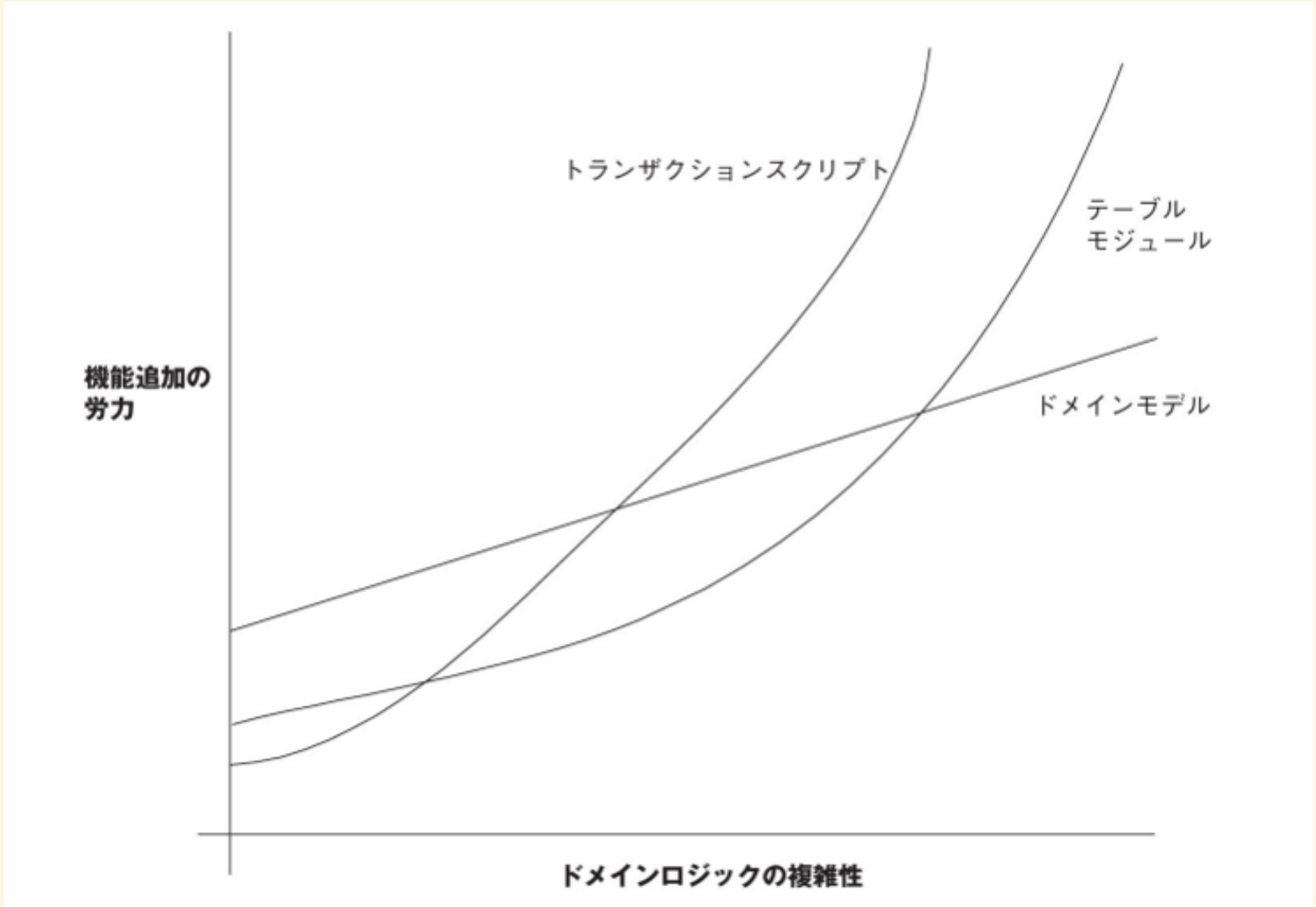
プログラミングにおける設計

- イメージしたものを実現するまでの見通しを立てる行為
- 一旦は完成した後も、ものが変わり続けるために、変化への強さが求められる

複雑性と設計

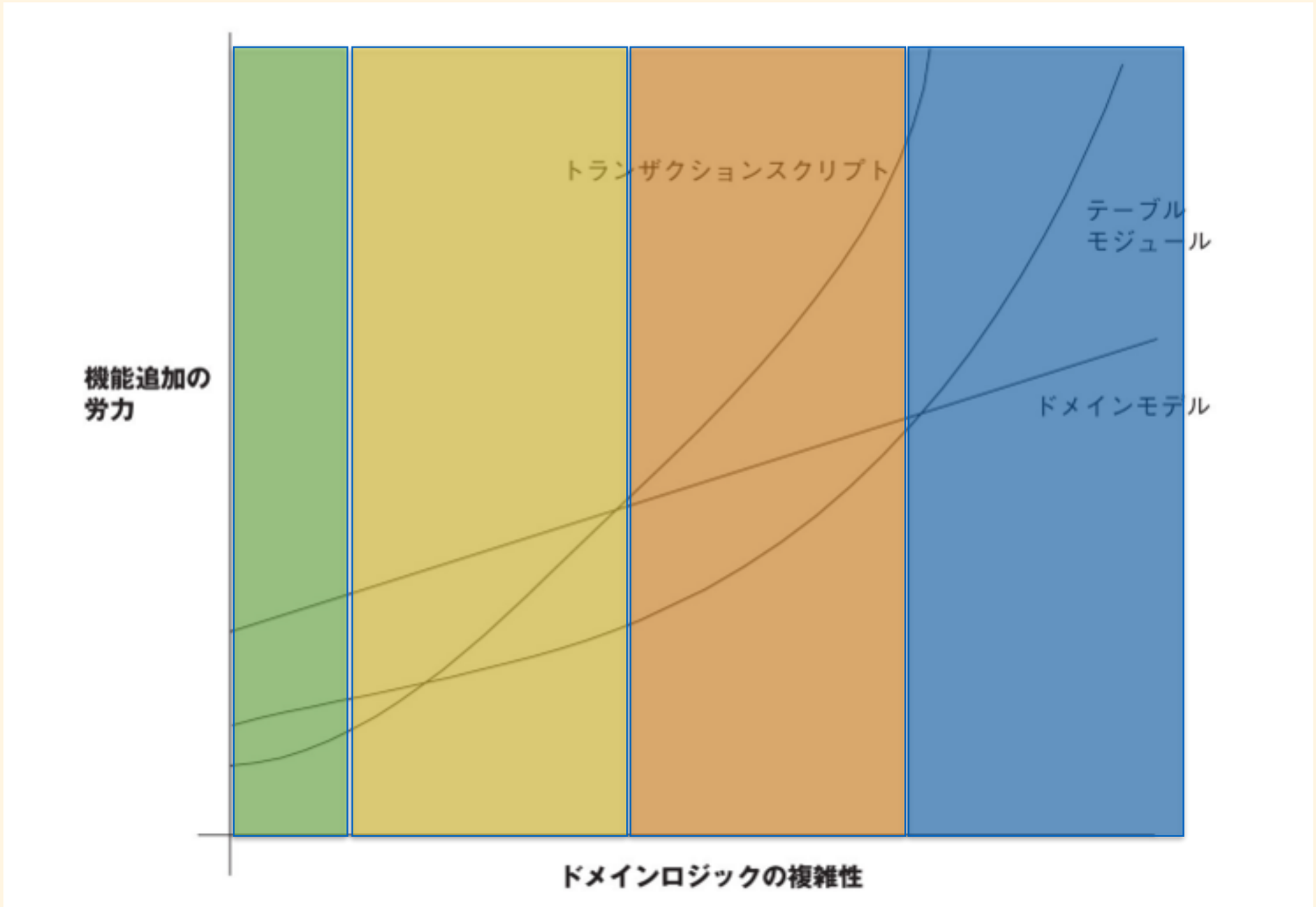
設計は規模や複雑度によって
適切なものが変わってきます

各設計手法の複雑性と労力のグラフの例

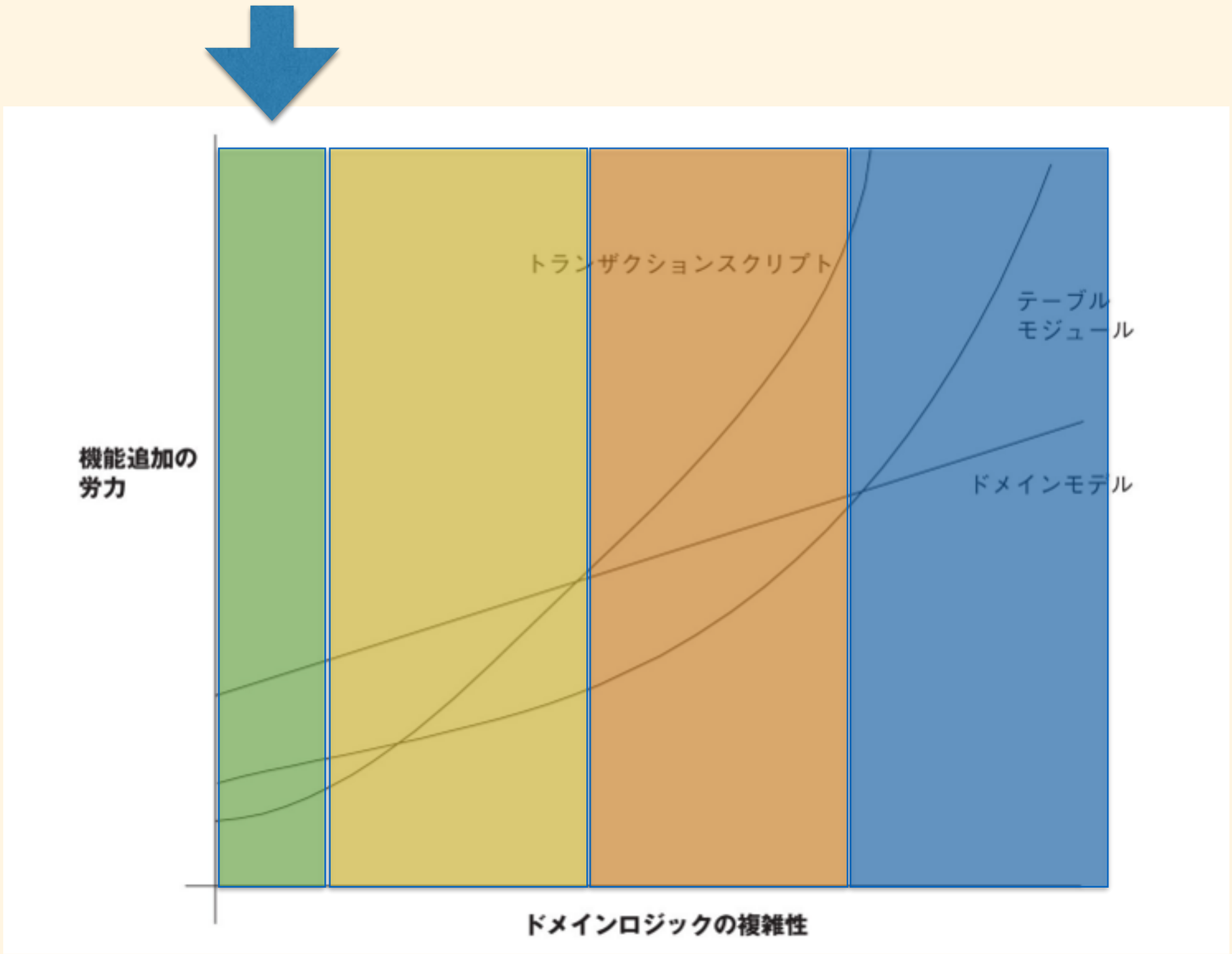


参照: Patterns of Enterprise Application Architecture

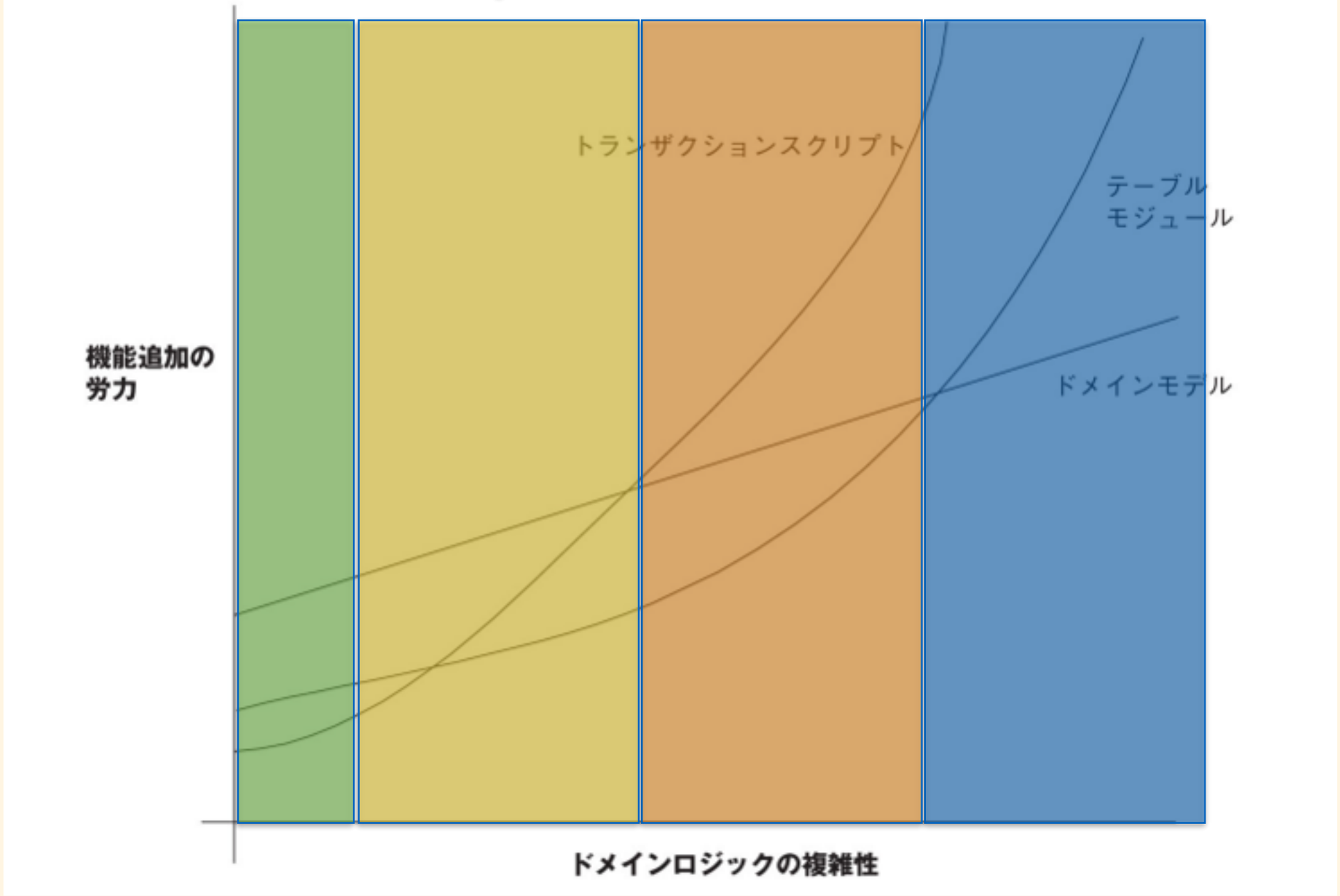
感覚ですが、こんなイメージ



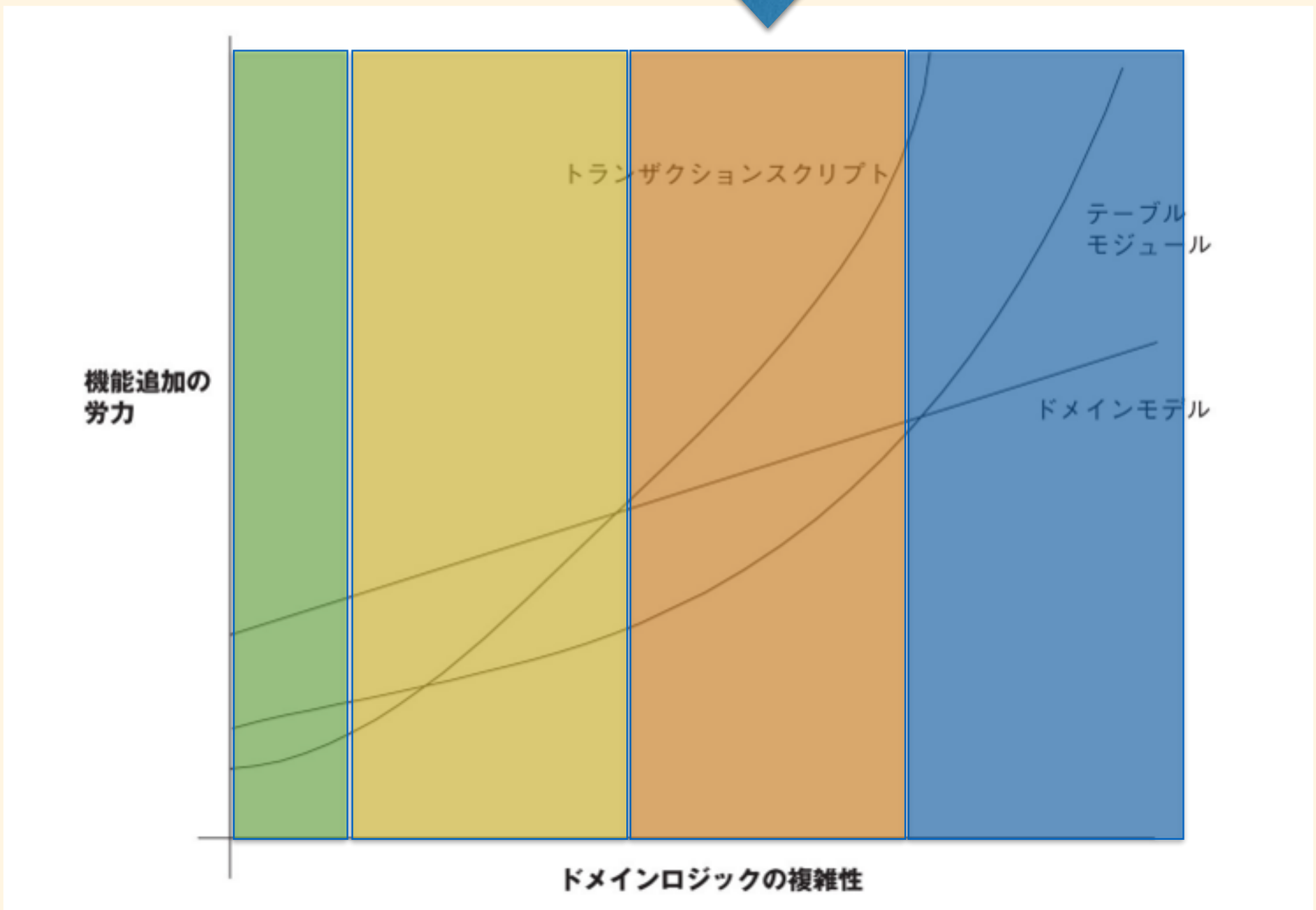
Hello, World レベル



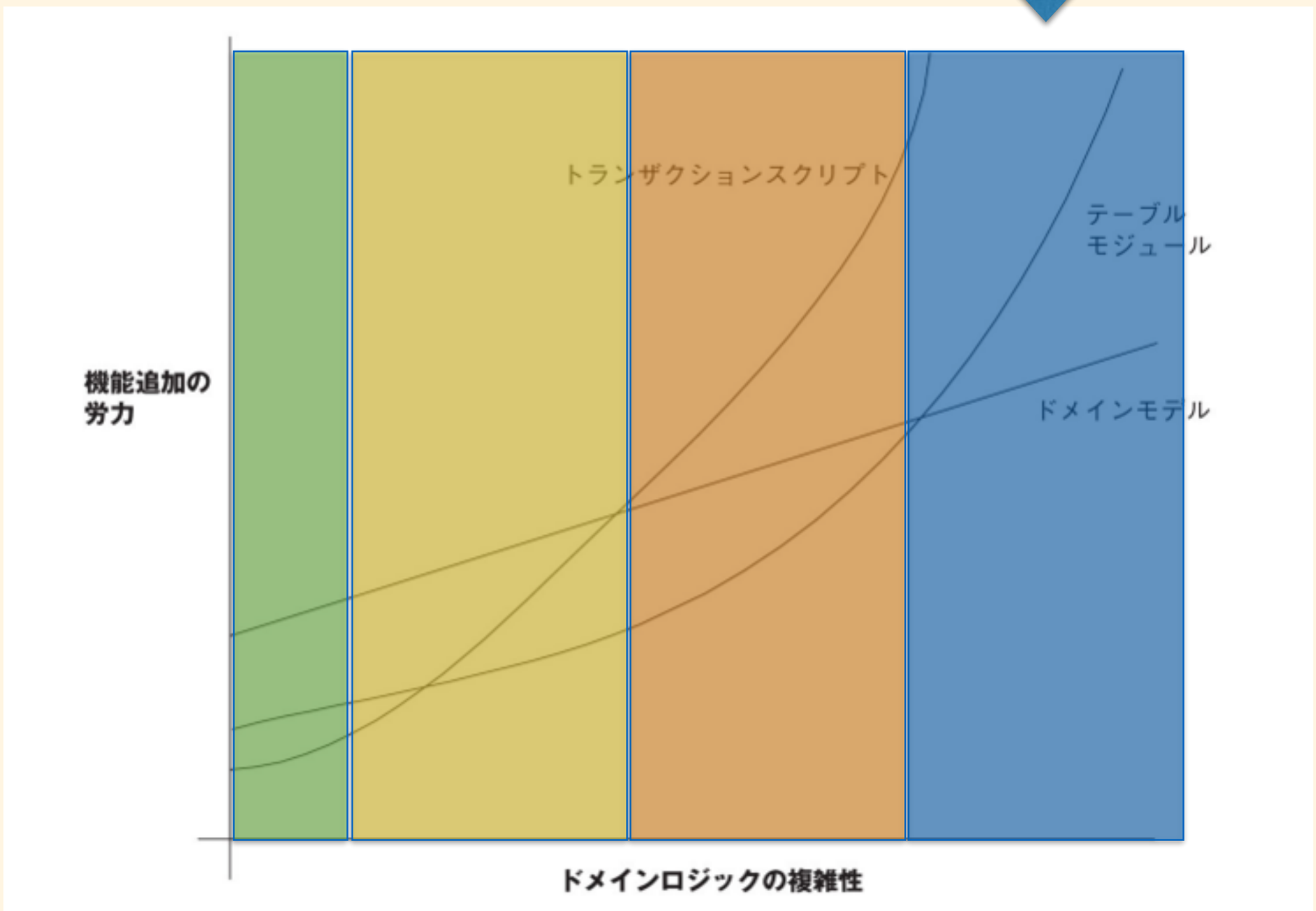
APIちょっと呼ぶだけ
特定の機能使うだけ

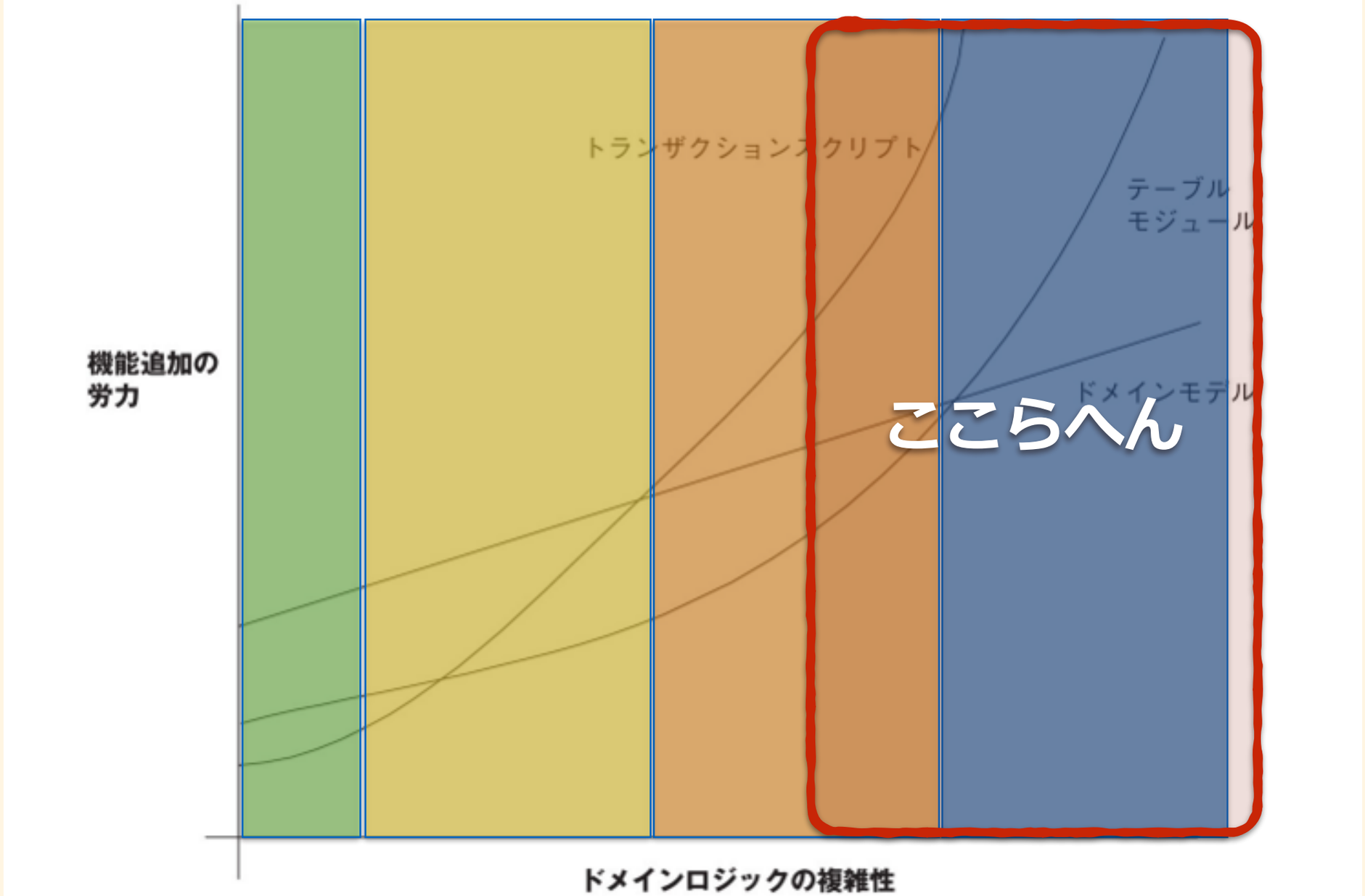


ある程度のコンテキストの違った
API群があったり
それをキャッシュしたり



多くのコンテキストの違ったAPI群があったり
それをキャッシュしたり





感覚ですが、この辺でAndroidをするなら

適しているのだろうなと思う

設計の話をします

MVPとDDD

結論から言うと

自分はMVPとDDDを

選んだ

なぜMVPか

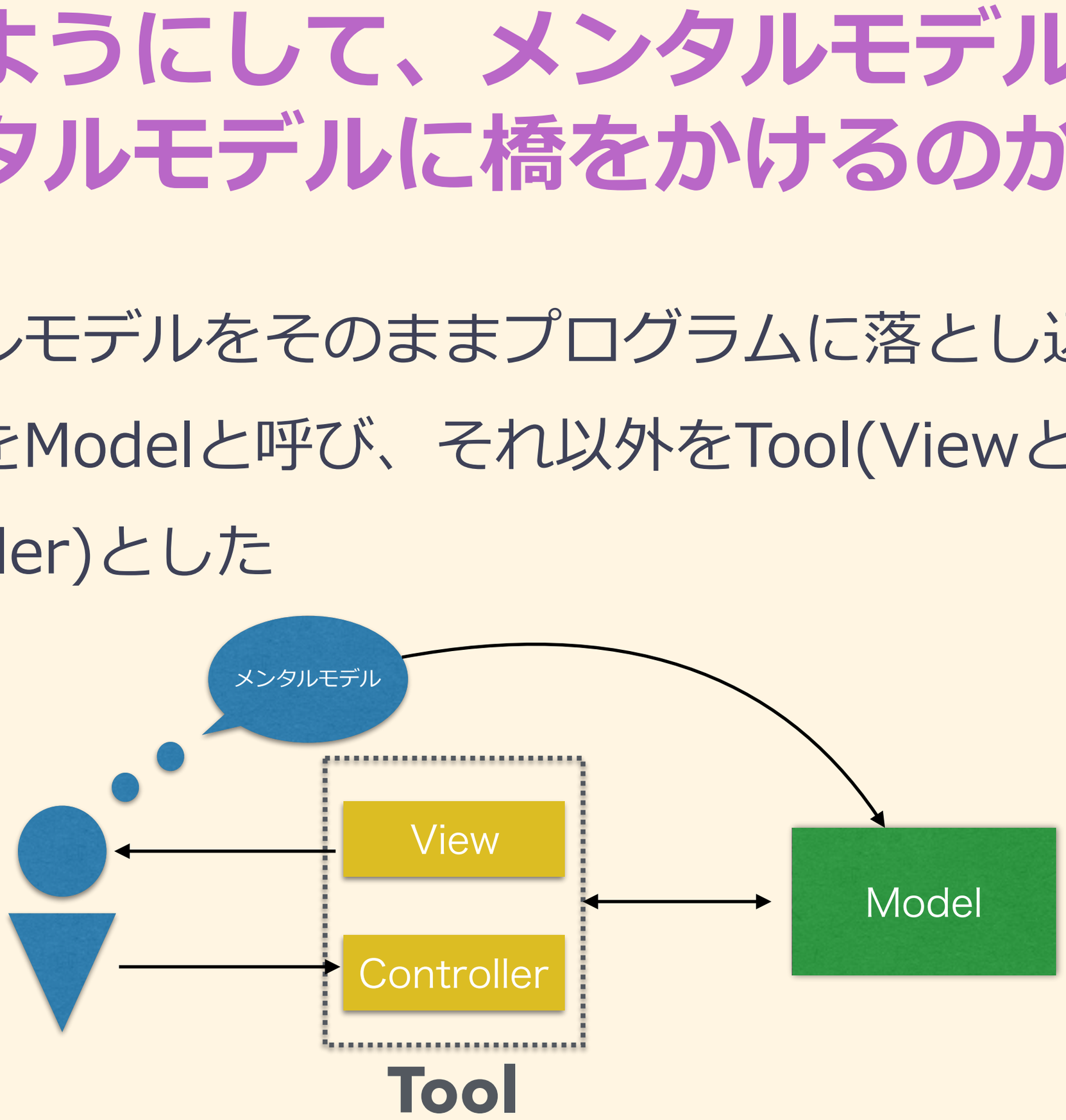
MVCの目的

- 「MVCの目的は、ユーザーのメンタルモデルとコンピュータのデジタルモデルに橋をかけることである」–Trygve Reenskaug

- MVPはMVCの派生なので同様の目的を引き継いでいる

どのようにして、メンタルモデルとデジタルモデルに橋をかけるのか？

メンタルモデルをそのままプログラムに落とし込んだものをModelと呼び、それ以外をTool(Viewと Controller)とした



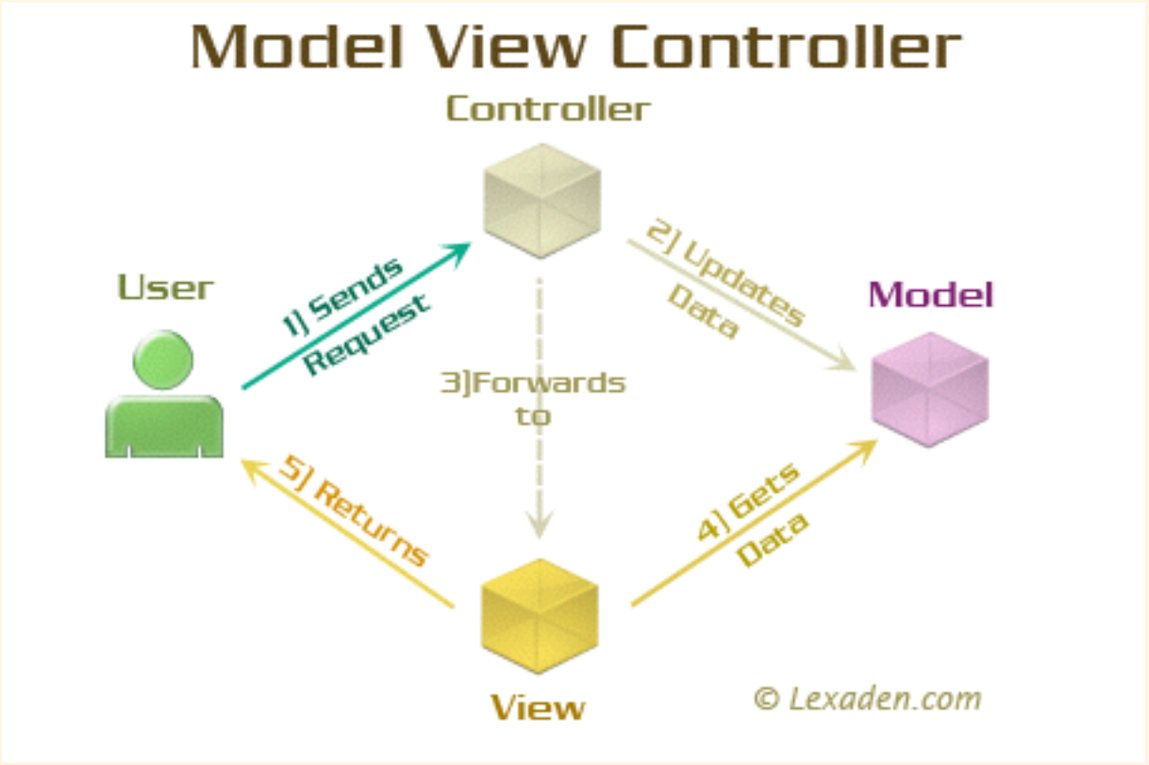
MVCは
イメージしたものを
プログラムに落とし込むための
フレームワークである
そのためMV○を選んだ

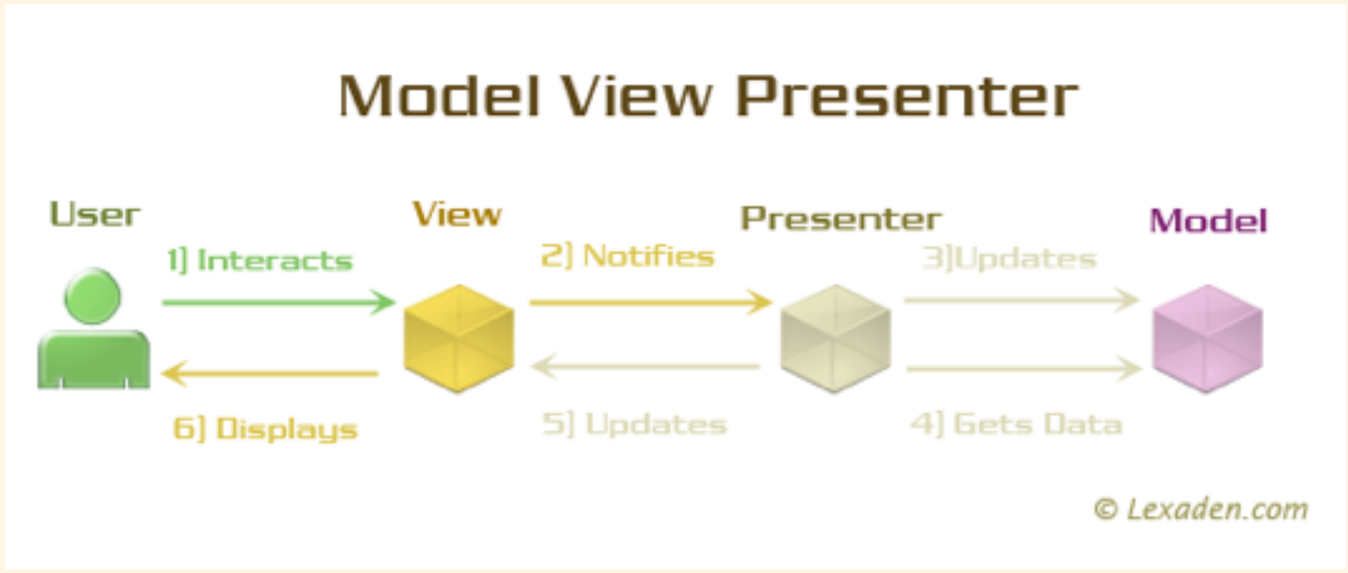
MV○の中でも

MVPである理由

MVCとMVPの違い

- フローが違う
- Viewが必ずController(Presenter)を通るかどうか





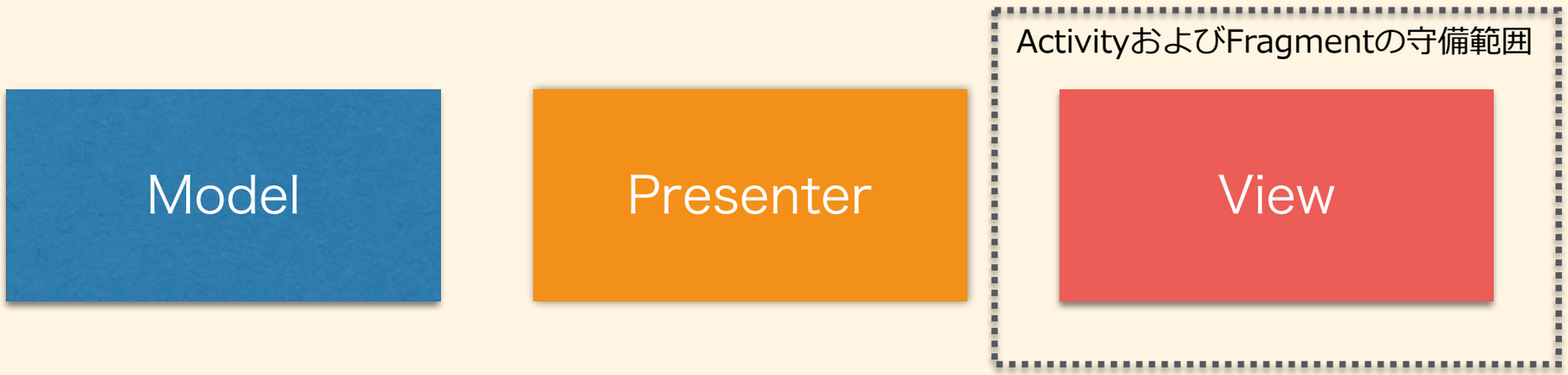
参照: http://stackoverflow.com/questions/2056/what-are-mvp-and-mvc-and-what-is-the-difference

AndroidにおけるActivityとFragmentがFatになりやすい問題

- ActivityおよびFragmentはViewでありControllerである

AndroidにおけるActivityとFragmentがFatになりやすい問題

- **MVP**を適用し、ActivityおよびFragmentをViewとして、これらが持ってたイベント管理責務をPresenterに外出しするとActivityとFragmentがViewに専念できるようになる
- (ActivityとFragmentからイベント管理を取り除いてみたらMVPになりました、という理解でもまぁOK)



MVPを適用した理由

- MVCと同様、「メンタルモデルをいかにデジタルモデルに落とし込むか」を考えたフレームワークであるから

- AndroidのActivityとFragmentのFat問題のソリューションとして適していたから

なぜDDDか

MVC(P)ではモデルに触れていない

- 「メンタルモデルをデジタルモデルに落とし込む」重要性は説いてるが、その手法については触れていない

- そこに触れているのがDDD

以前はレイヤードな
クリーンアーキテクチャを採用していた

- はじめはすごく見通しが良かった

- 様々なコンテキストが絡むようになって、プログラムの見通しが悪くなってきた

- 度重なる仕様変更により正しい仕様をプログラムで表現する難易度が上がってきていた

DDDを選ぶ理由

- **DDDは言葉と実装を一致させる設計手法**
- 言葉と実装を一致させる意識がないと、そこにねじれが出てくる
- 規模が大きくなるにつれて、ねじれは大きくなり、変更が大きく、バグが生まれやすくなる

言語と実装を一致させるとは？

例えばフィードの投稿機能を考える

- データモデル
  - Feed フィードクラス
    - pojo になる
      - getter / setter
      - Gsonでシリアライズしたりする

- 「投稿を編集する」、「コメントを追加する」、「コメントを削除する」機能を実装する
  - 各ActivityやFragmentに書いたり
  - そこに直書きしないとしても、各画面寄りの場所に実装することになる

=> **Feedの振る舞いの実装が散らばる**

- ドメインモデル
  - データだけでなく振る舞いもモデルに含む
  - Feedクラス
    - edit, addComment, removeComment

=> **言語と実装が一致する**

DDDを適用した理由

- 言語と実装を一致させるための手法でモデルがシンプルになりやすい
- 新しい機能の実装時や既存機能の修正時、実装箇所が明確になる

各レイヤーと各クラスの説明

Presentation

Presenter

View

双方向binding

ViewModel

Converter(entity/vo ⇄ viewmodel)

Application

UseCase

Presenterからコマンドを非同期スレッドで実行し、EventBusで結果を返す
実行内容はentityやdomain serviceのメソッド呼び出したり、調整したりするだけ。

Domain

Entity

Value Object

Domain Service

Repository

Converter(dto ⇄ entity/vo)

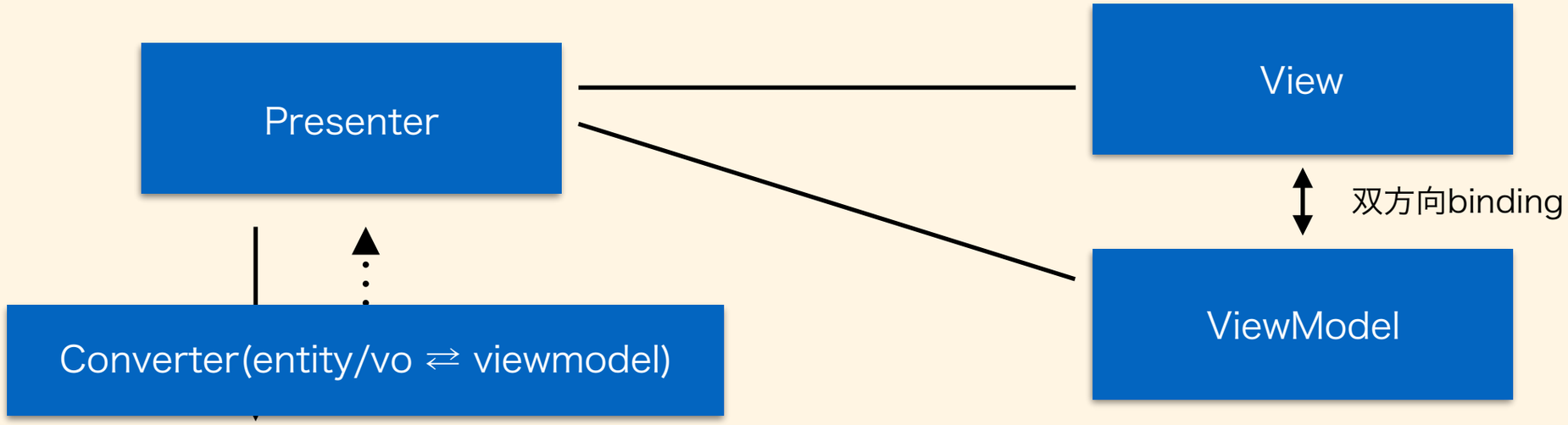
Infra

API

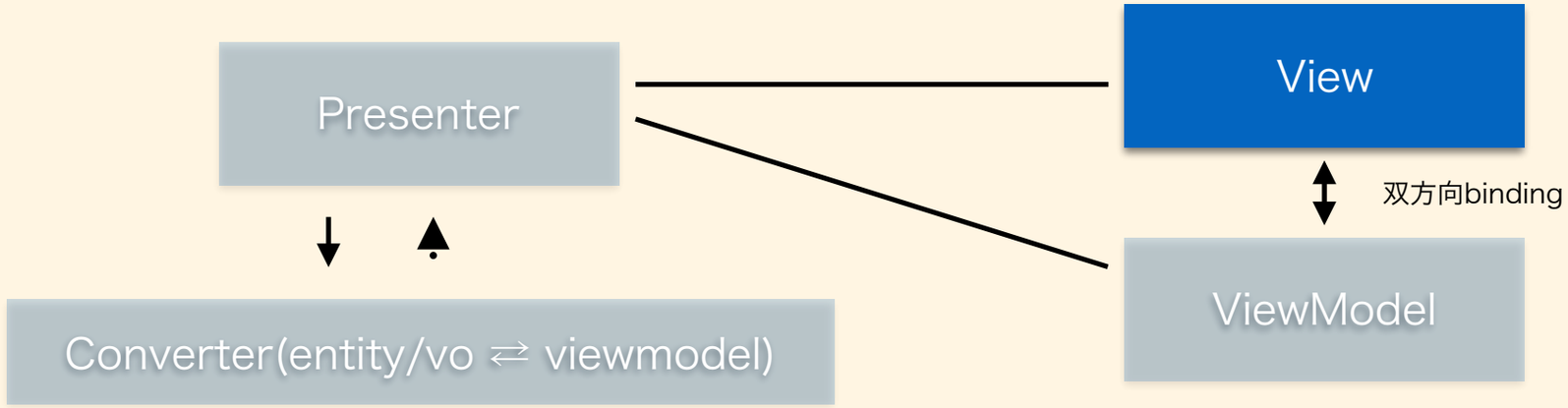
Shared Preference

local db

Presentation Layer



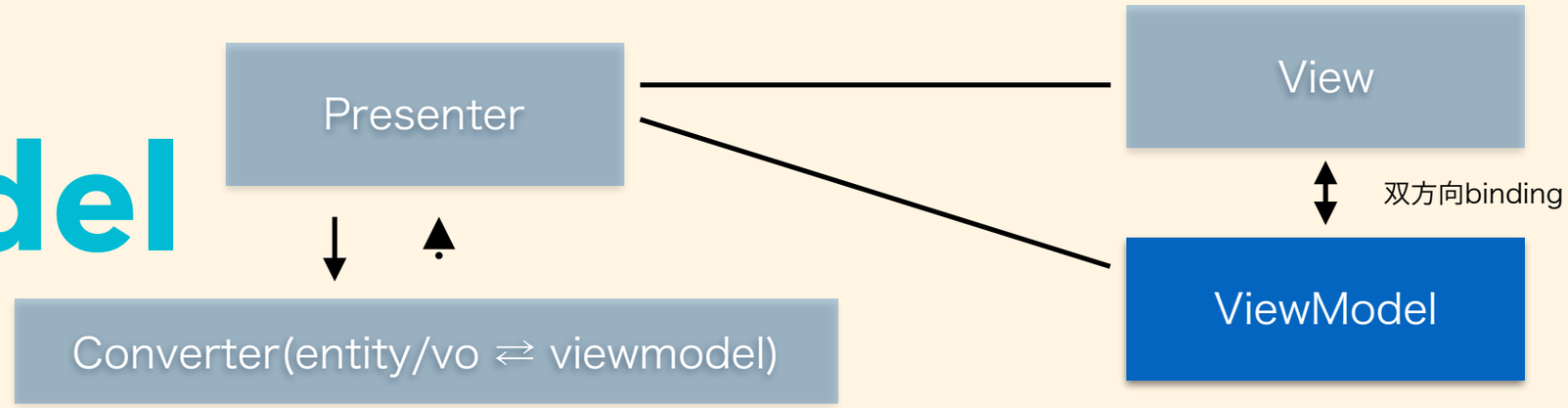
View



- 画面に表示するもの

- ○○View / ○○Layout / Activity / Fragment

ViewModel

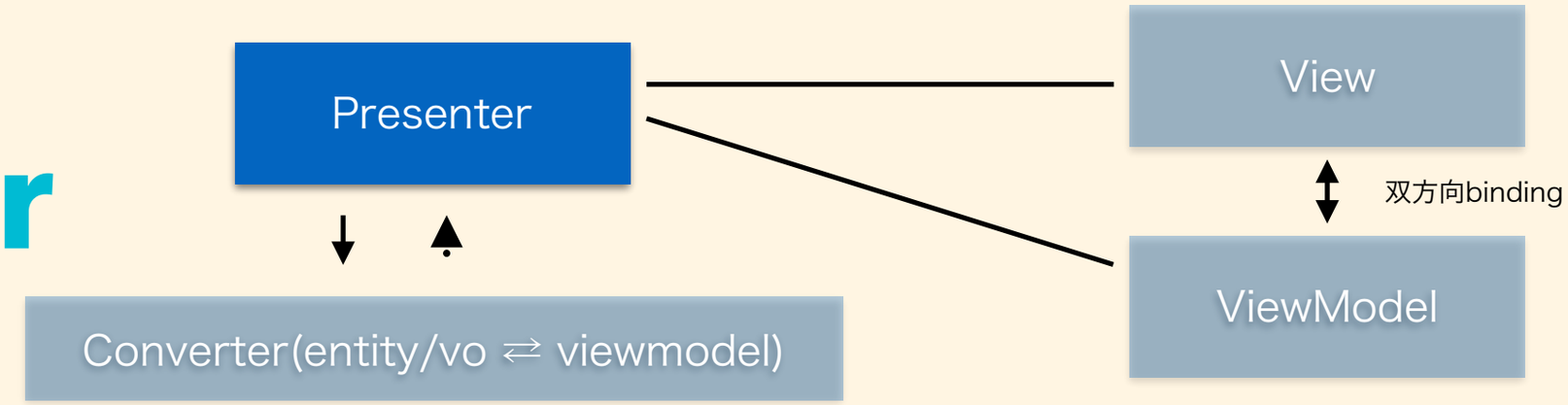


- 画面に表示用のモデル

- 画面に表示されている情報を持つ

- Viewと双方向Bindingする

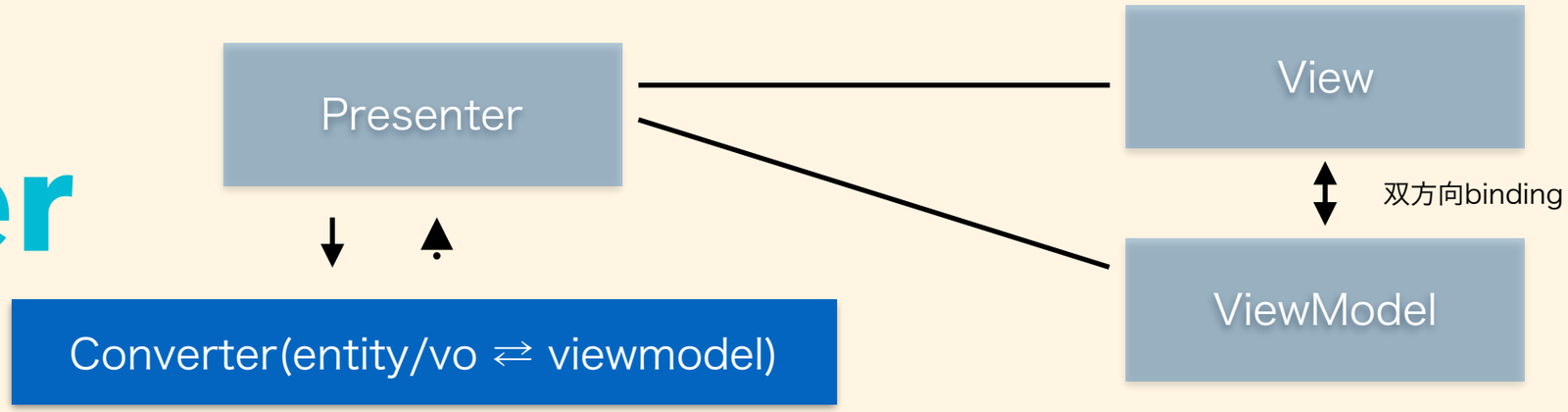
Presenter



- イベント管理

- Viewでおこるイベントは全て Presenterに移譲する

Converter



- 下の層からくるEntityやValue ObjectをView Modelに変換する

- 下の層に送るときは逆にEntityやValue Objectに変換する

- 愚直なコードになりがち

Application Layer

UseCase

- DDDで言うところのApplication Service
- ドメインではなくApplicationのために必要な動きをする
- ドメインへのコマンドを別スレッドで行い、結果をUI スレッドへ返す

Domain Layer

Entity

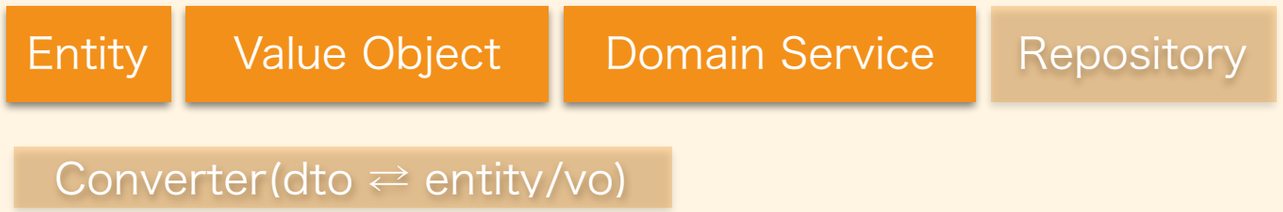
Value Object

Domain Service

Repository

Converter(dto ⇄ entity/vo)

Entity
Value Object
Domain Service



- モデルそのもの

- EntityとValue Objectは属性と振る舞いをモデリングしたもの

- Entityには識別する必要があるもの(人とか)

- Value Objectは識別する必要がないもの

- Domain Serviceは何かオブジェクトするのが不自然な時に振る舞いのみをモデリングしたもの

Repository

Entity | Value Object | Domain Service | Repository

Converter(dto ⇄ entity/vo)

- Entityのライフサイクル管理(永続化や問い合わせなど)
- APIの都合でValueObjectとってくることもある

Converter

Entity | Value Object | Domain Service | Repository

Converter(dto ⇄ entity/vo)

- Infra層などからくる、DTO(Data transfer object)を Entityや Value Objectに変換する
- またその逆もする

Infra Layer

API

Shared Preference

local db

Infra Layer

- 永続化基盤 (ネットワーク基盤など)などをここに置く
- それ以外にも純粋な技術的サービスを置く (メッセージング基盤など)

パッケージ図参考

```
|- common
|- infra
|    |- converter
|- usecase
|    |- (コンテキスト名)
|        |- HogeUseCase
|- domain
|    |- (コンテキスト名)
|        |- model // ここにentityとvalueobjectとdomain serviceをつっこむ
|        |   |- Hoge
|        |   |- FugaService
|        |- repository
|        |   |- HogeRepository
|        |- conveter
|            |- Converter
|- presentation
|    |- (コンテキスト名)
|        |- HogeActivity
|        |- HogeFragment
|        |- ViewModel
|        |- BindingAdapters
|        |- HogePresenter
|        |- HogeView
|        |- Conveter
```

まとめ

まとめ

- 設計手法のコスパは複雑性に依存している
- 複雑性が大きい場合は、MVPとDDDが適しているだろうと思った
  - MVPは複雑になりがちなActivity/Fragmentからイベント管理責務を排除する
  - DDDは言語と実装を一致させる