プログラミング言語とシステムデザイン

Clojure, Java, デザインパターン, DDD, Clean Architecture

2021/12/18 株式会社ユーザベース 矢野勉

提供

UZABASE

株式会社ユーザベース

誰？

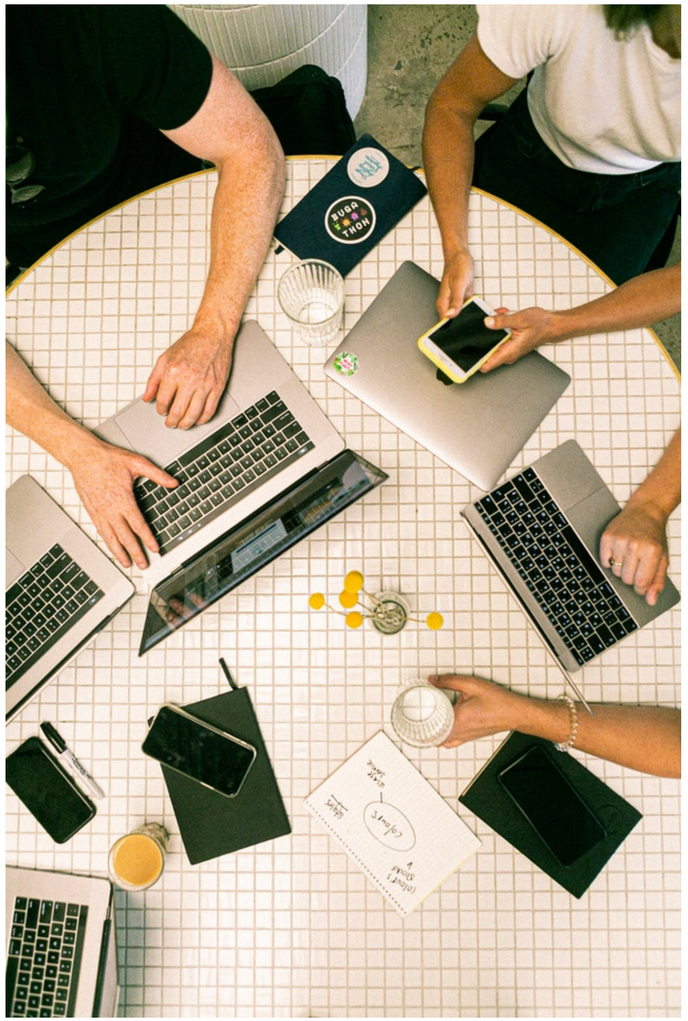
- 名前： 矢野勉
- @t_yano
- https://boxofpapers.hatenablog.com/
- 株式会社ユーザベース・技術部門のフェローをやってます
  （フェローというのはユーザベースの正式な役職名で、別に名誉職ではないです。仕事してます）
- プログラミング言語Clojureのコントリビュータ
  コンパイラにパッチ送ったりしてます（なかなかマージされない...）

モデリング議論楽しいですよね

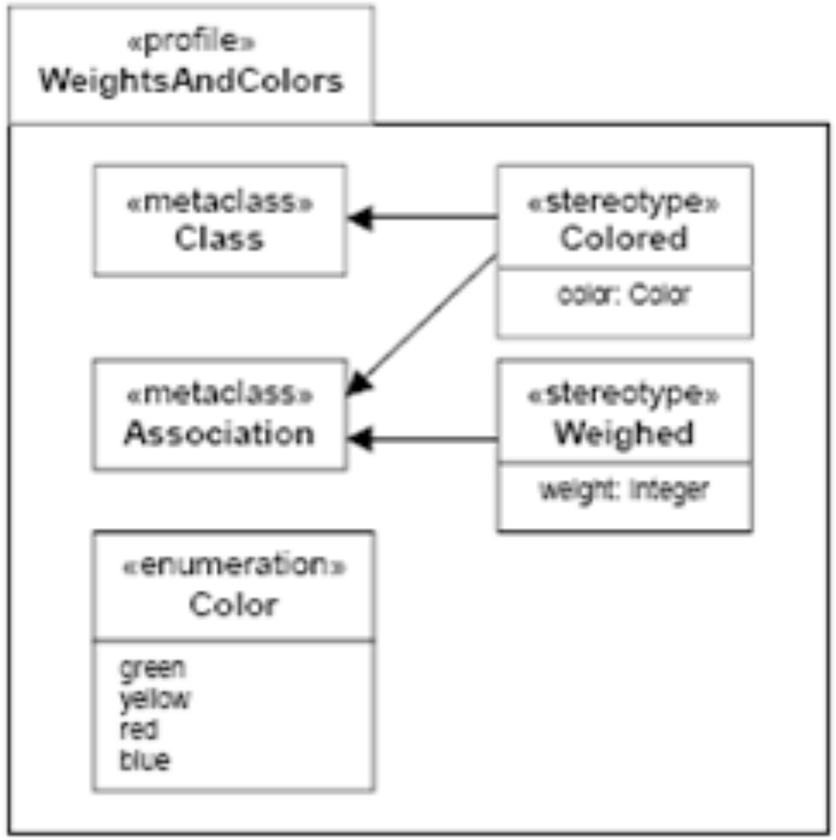
けっこう白熱しますよね

これはこういうクラスに分割しよう

これはドメインかな、ドメインサービスかな



なんのためのクラス分け？



夢中になると、なんのためにクラス分けしようとしてるのか、思考が止まってしまう

デザパタ地獄

なんでもかんでも、なんらかのデザインパターンを適用しようとする麻疹

AbstractFactoryパターンがあるからnewは禁止ですか？

シンプルな委譲で済むことに、ChainOfResponsibilityパターン必要ですか？

なんでも過剰になっていく

XML最高 → EJB2

XML-RPC →もっと機能増やすぜ → SOAP

REST最高！→ RESTful+HATEOAS

トレードオフ

設計にはトレードオフがつきもの

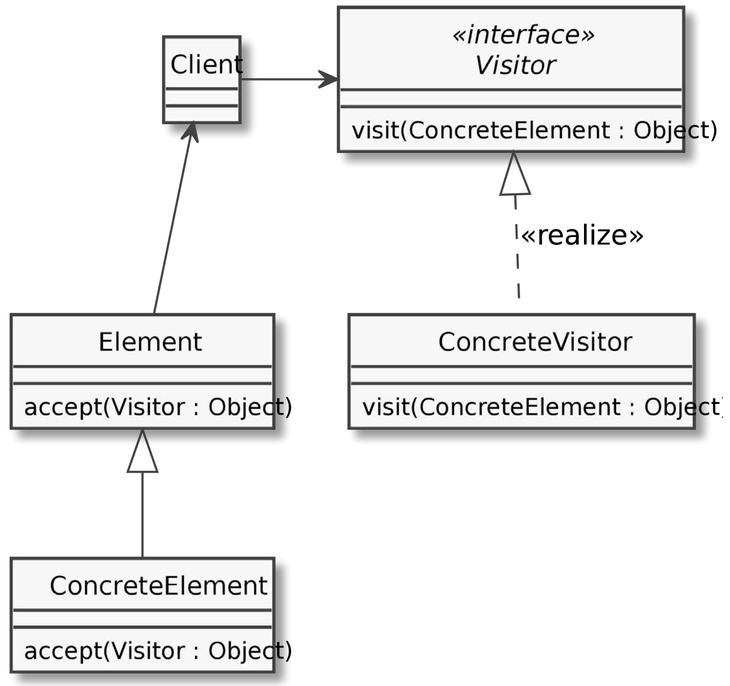
やりたいことに対して複雑さが高過ぎては意味がない

パターンは言語によっては消滅してしまう

デザインパターンはトレードオフで選択するもの

Visitorパターン

Visitorオブジェクトを定義することで、階層状のノードを潜りながら、ノードに対する操作を実行していくデザインパターン



- Visitorはクロージャ関数として書けるし、ノードを潜る操作は再帰処理として書けます。

クロージャ関数があって、再帰処理をサポートしている言語で、
このパターン必要ですか？

デザインパターンには、適用条件があって、その条件がなくなってしまったら、パターンごと必要なくなる

条件が変わったら消えてしまうものがある。

でも条件が変わっても消えないものもある。

その消えないものの一つが

設計理論

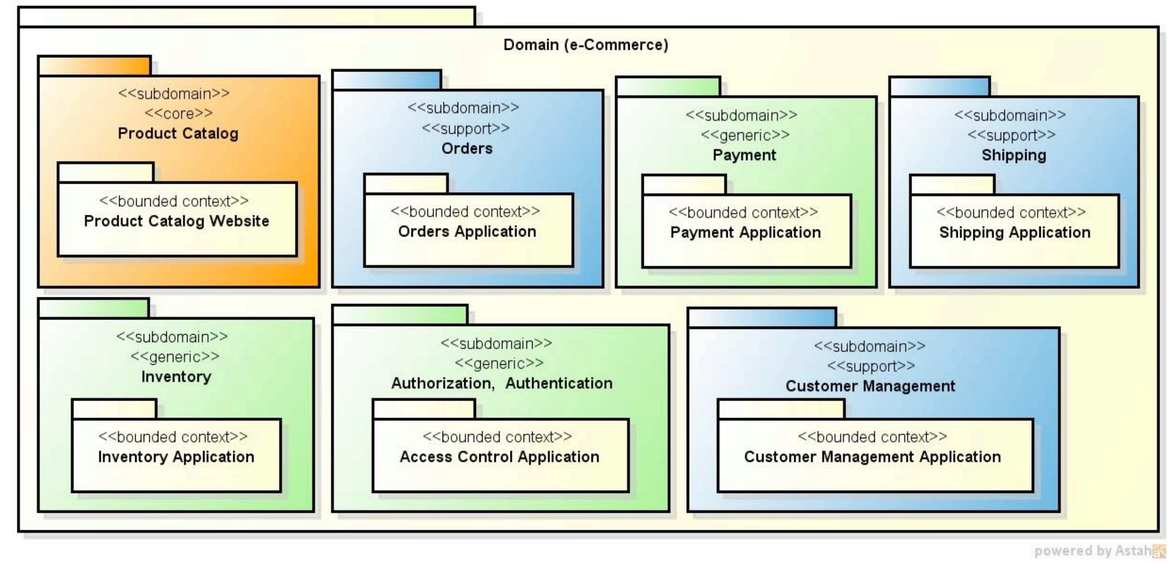
で

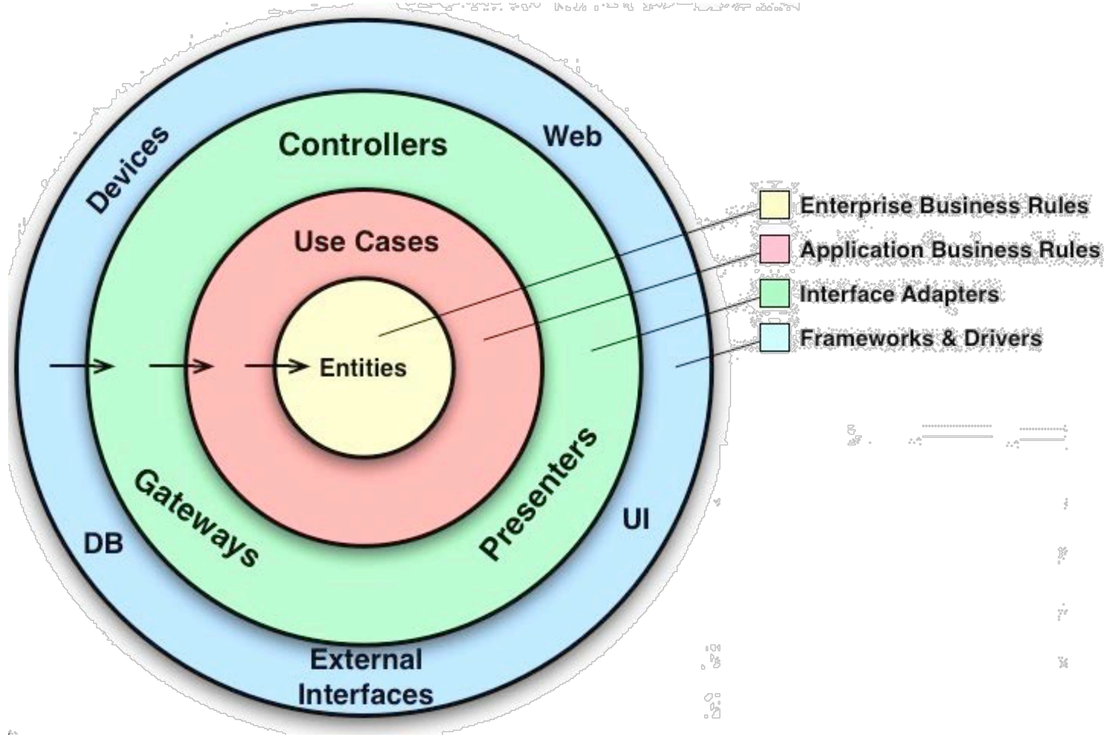
それを適用したものが

アーキテクチャ

なんじゃないかと思います

DDD と Clean Architecture





Clojureというプログラミング言語を肴に、
DDDやClean Architectureがどう見えてくるのかって話をしたいです

Clojureってどんな言語？

LISP系のJVM上で動く言語

よくかっこ多過ぎと言われるアレ
（実際にはそんなに変わらんのだけど）

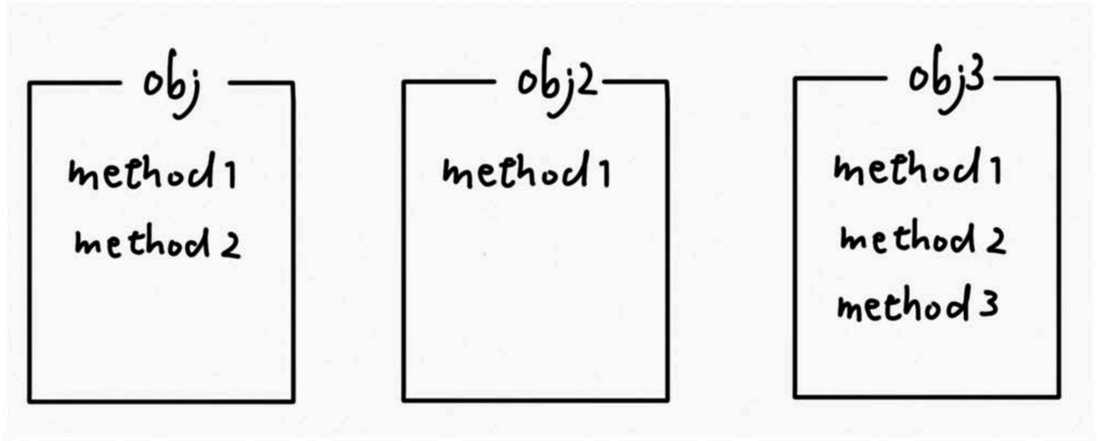
```
(defn- merge-configs* [a b]
  (merge/meta-merge (expand-ancestor-keys a b)
                    (expand-ancestor-keys b a)))

(defn merge-configs
  "Intelligently merge multiple configurations. Uses meta-merge and will merge
  configurations in order from left to right. Generic top-level keys are merged
  into more specific descendants, if the descendants exist."
  [& configs]
  (merge/unwrap-all (reduce merge-configs* {} configs)))

(defn- config-resource [path]
  (or (io/resource path)
      (io/resource (str path ".edn"))
      (io/resource (str path ".clj"))))

(defn resource
  "Return an record that represents a resource on the classpath, compatible with
  `clojure.java.io` functions like `reader`, `input-stream` and `as-url`. When
  printed, the record returns a string tagged with `#duct/resource`. This makes
  it more useful than a bare URL object when printing a configuration. If the
```

関数言語

- クラスはない
- 一つのデータに対してたくさんの関数

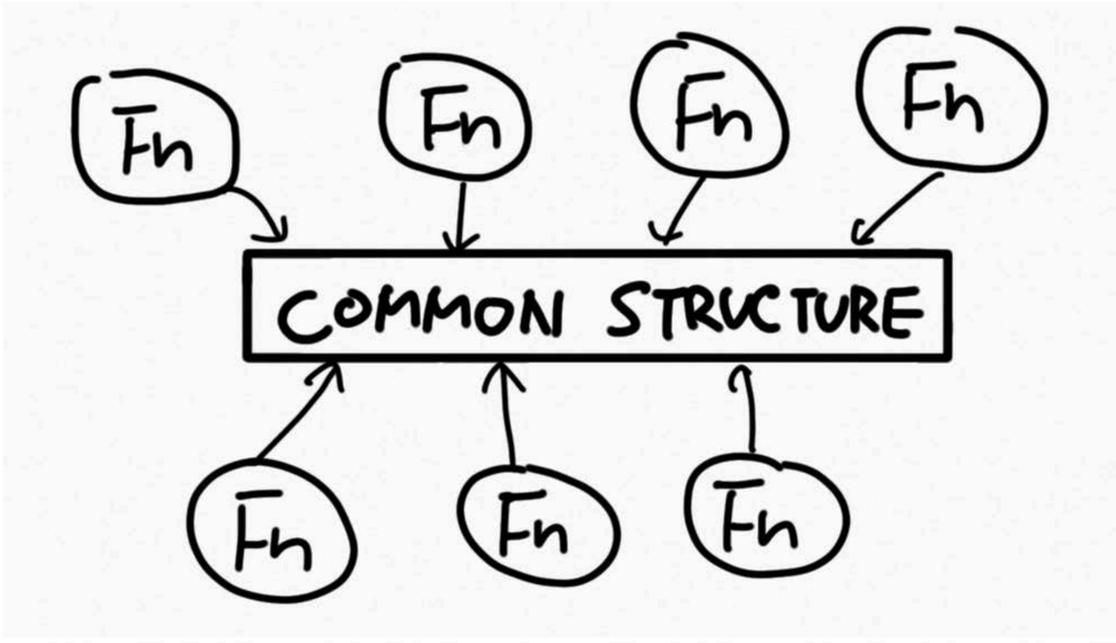
一般的なオブジェクト指向言語

データ構造にそれぞれ関数がある

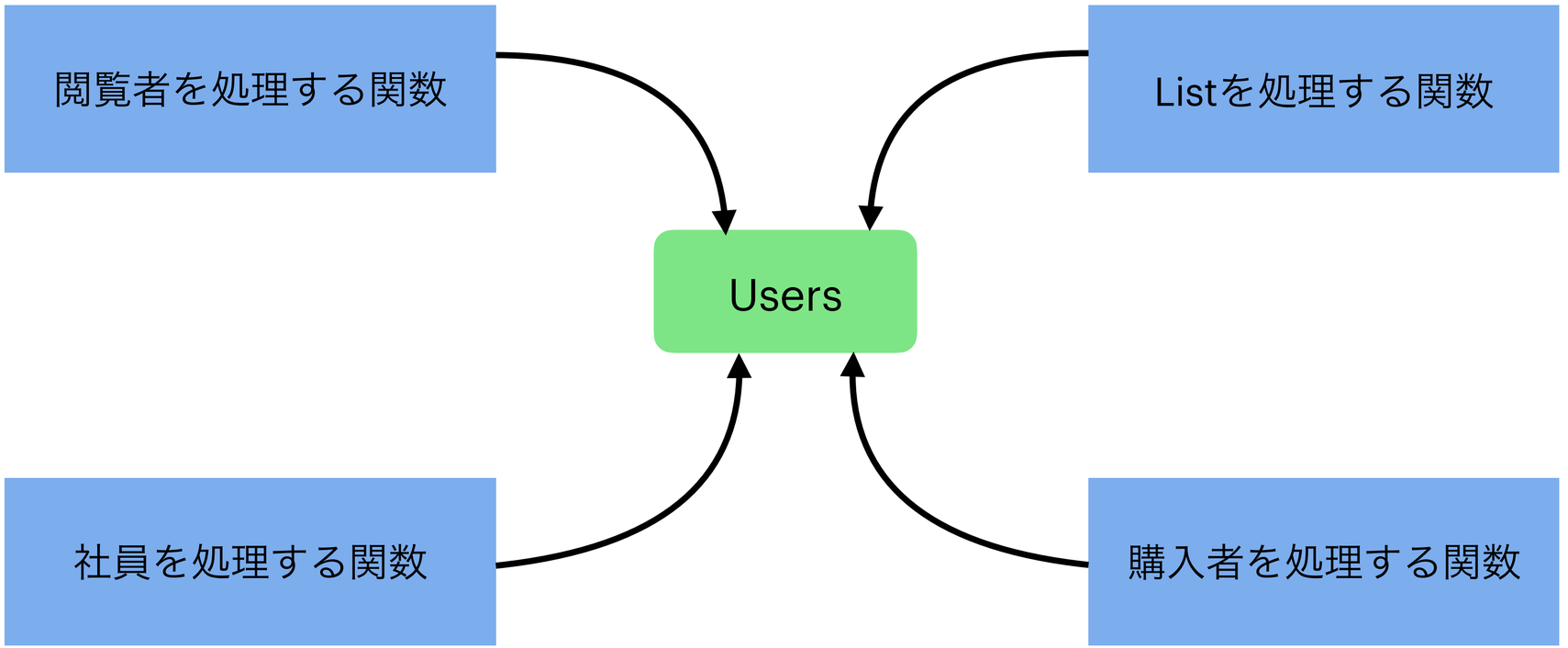


Clojure

一つのデータ構造を扱う100個の関数

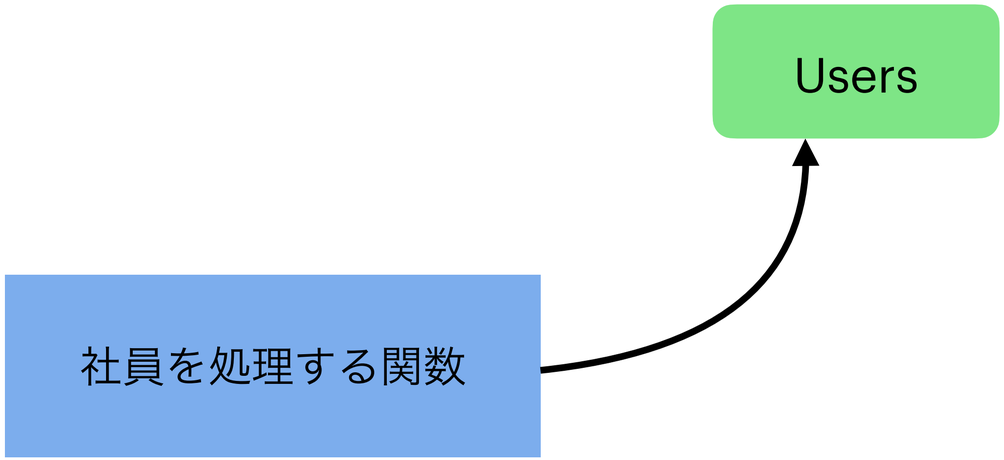


データと関数は独立している

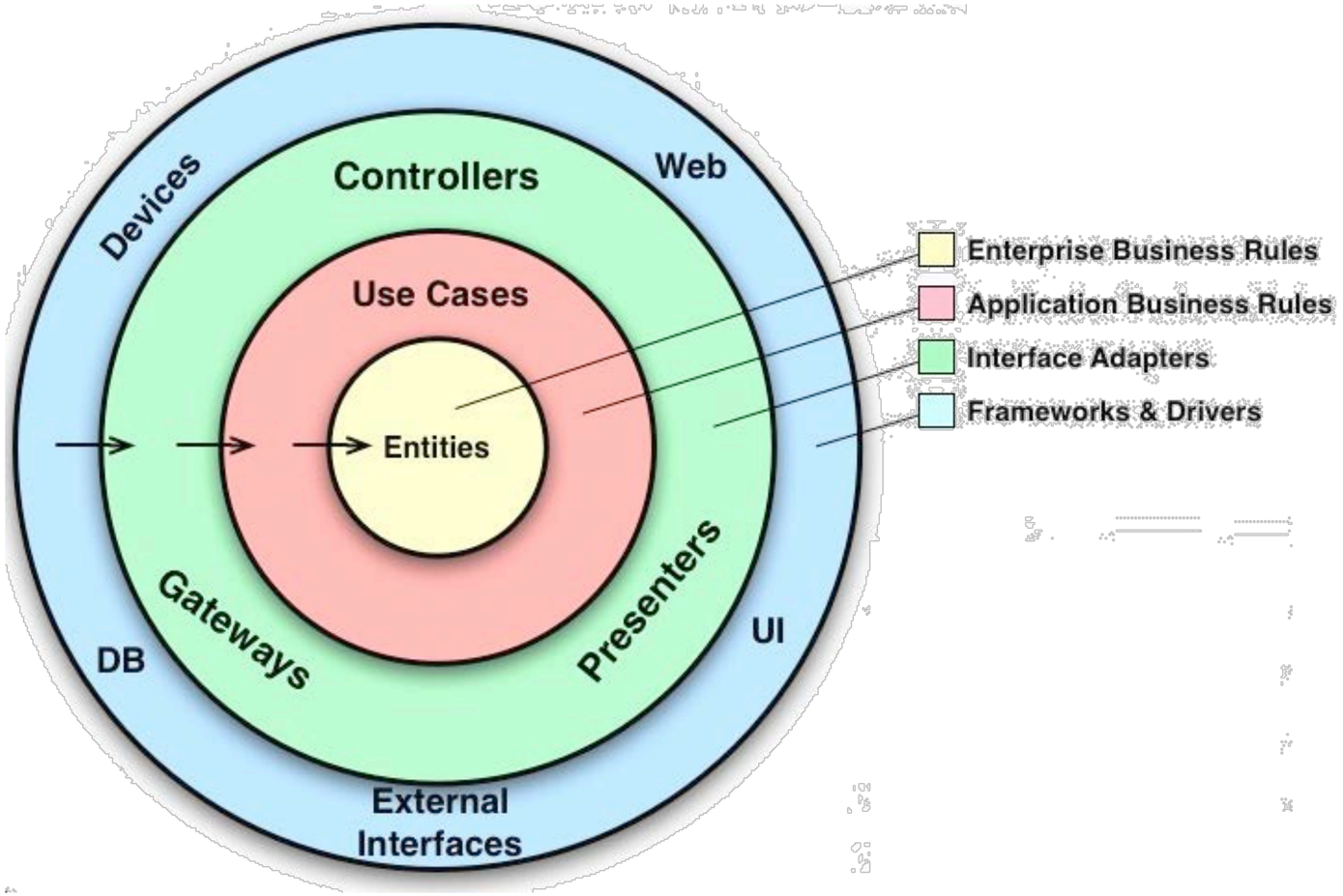


データの価値は、関数が決める

関数がそのデータを社員として扱うなら、それは社員データ



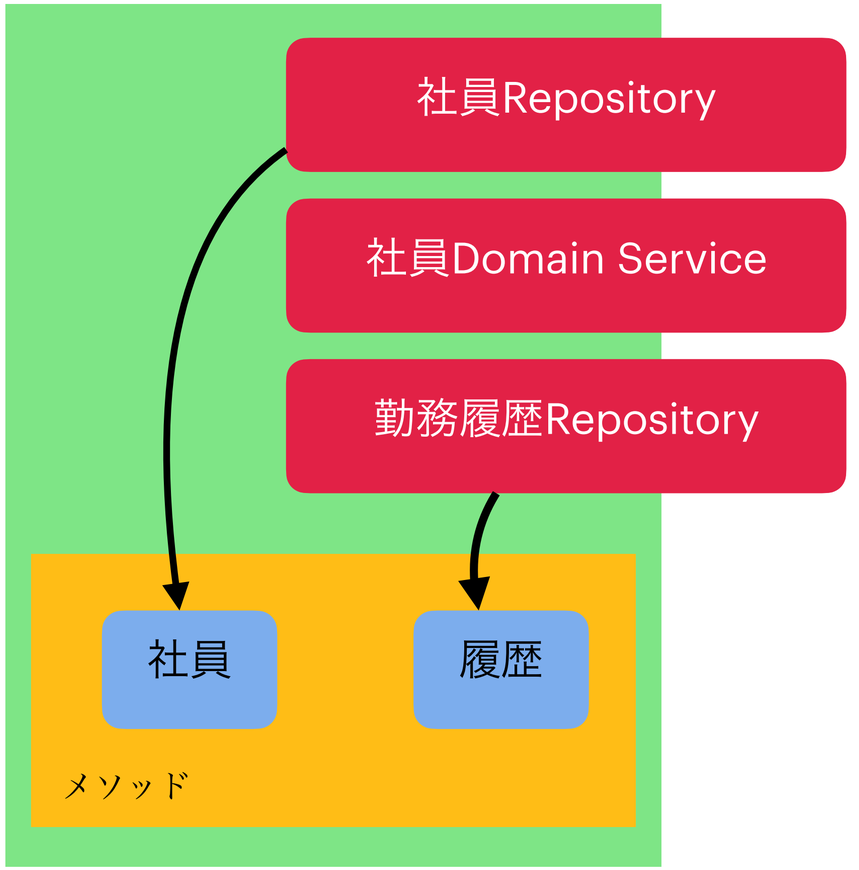
Clean Architectureのあの図



Usecaseってなんだ？

Usecaseオブジェクト？

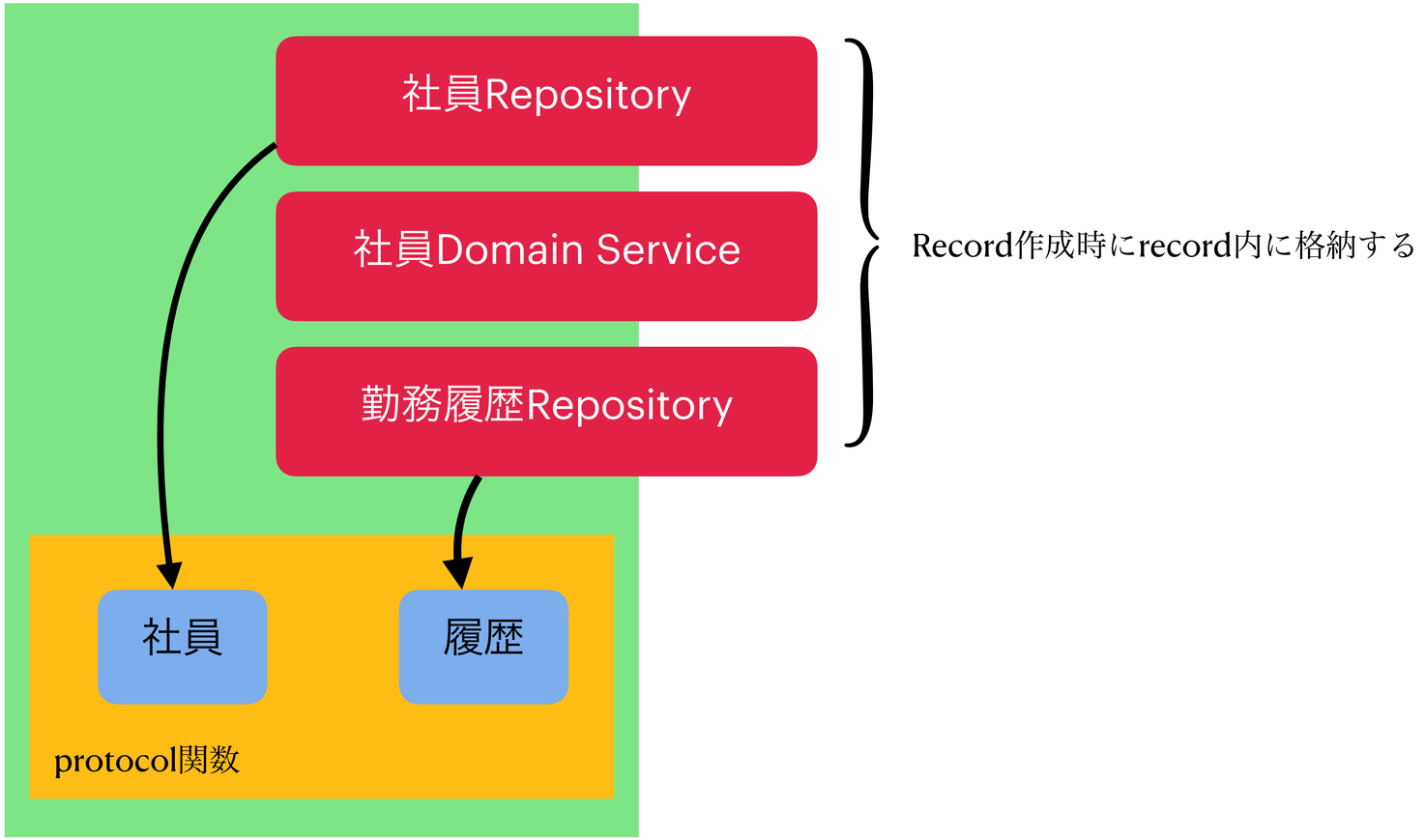
社員のある月のサラリーを求めるユースケース



Clojureでも似たような構造は作れるが...

RecordとProtocol

社員のある月のサラリーを求めるユースケースprotocolを実装したrecord
SalaryUsecase

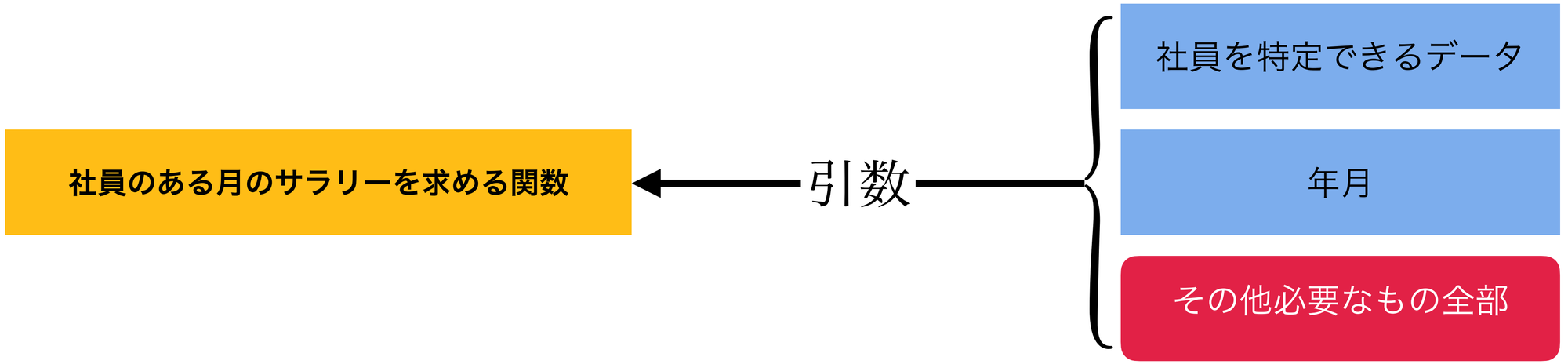


多くの関数言語で使われるポリシー

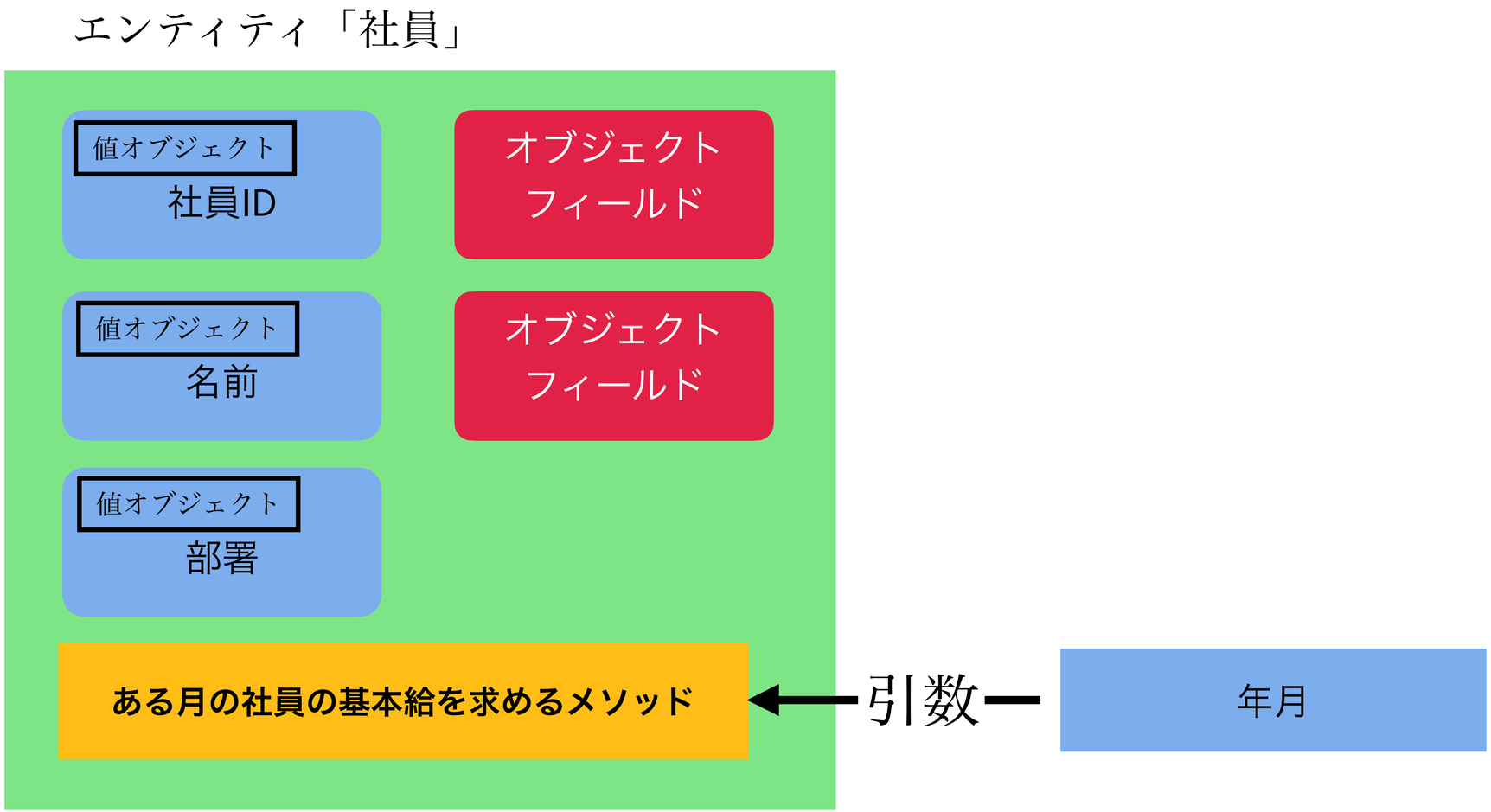
関数は、その引数にだけ依存すべき

- 関数はそれだけで実行できるべき
- テスト容易性
- REPLドリブン開発

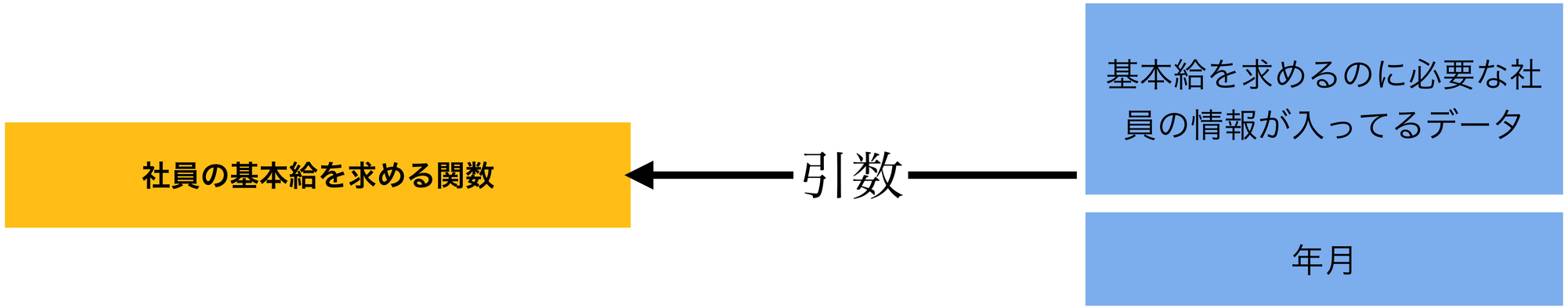
ただの関数でよくない？



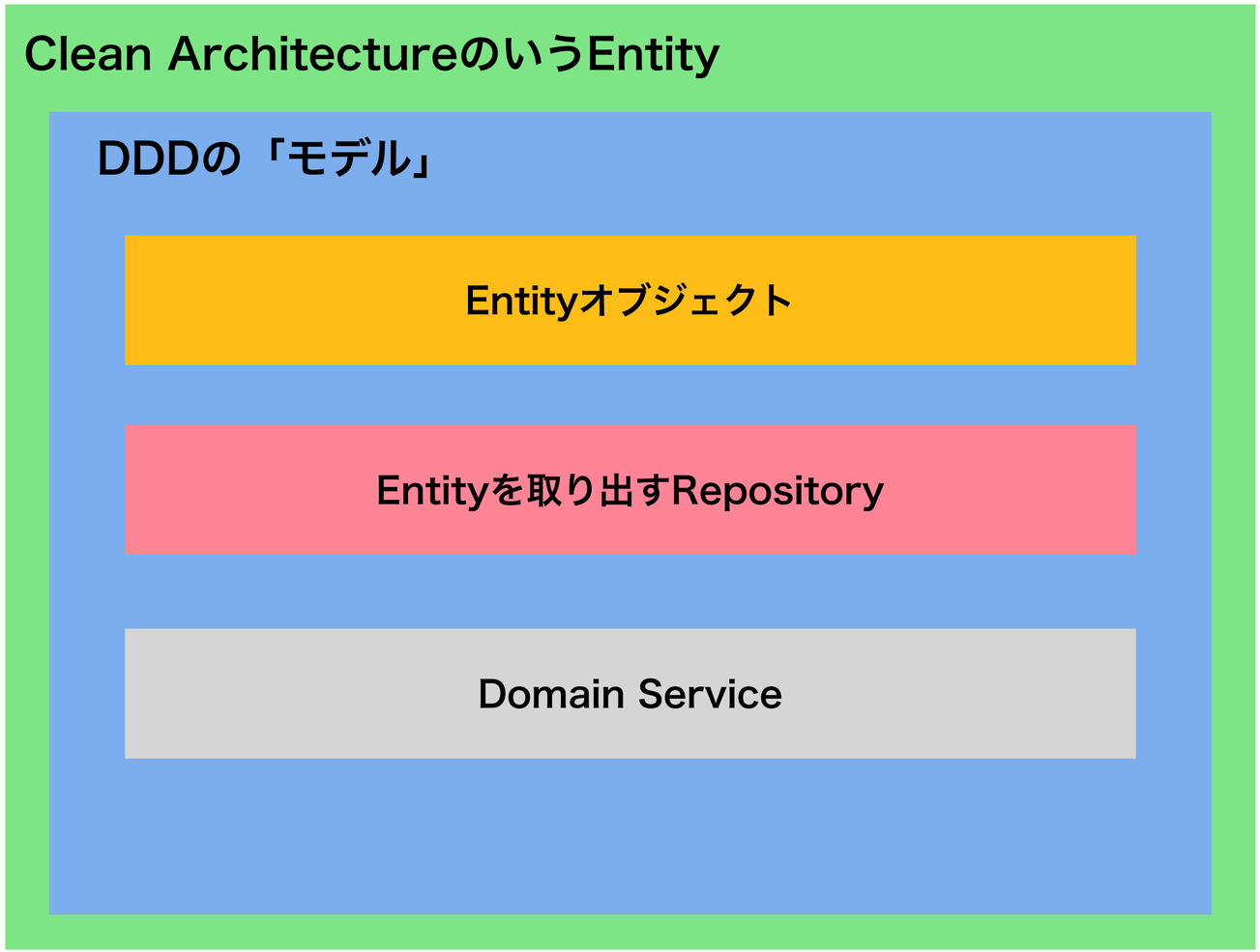
DDDのエンティティ


エンティティ「社員」
値オブジェクト
社員ID
オブジェクト
フィールド
値オブジェクト
名前
オブジェクト
フィールド
値オブジェクト
部署
ある月の社員の基本給を求めるメソッド
引数
年月

関数でよくない？

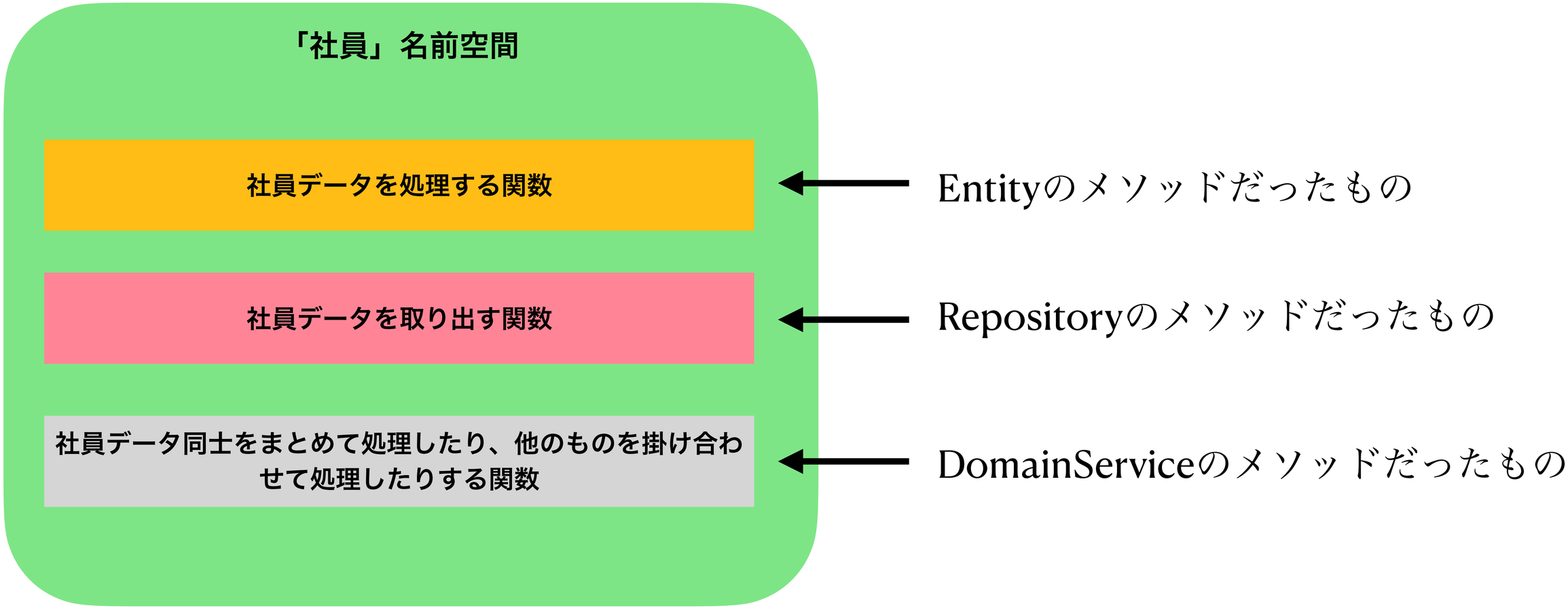


Clean ArchitectureのEntityと DDDのEntity



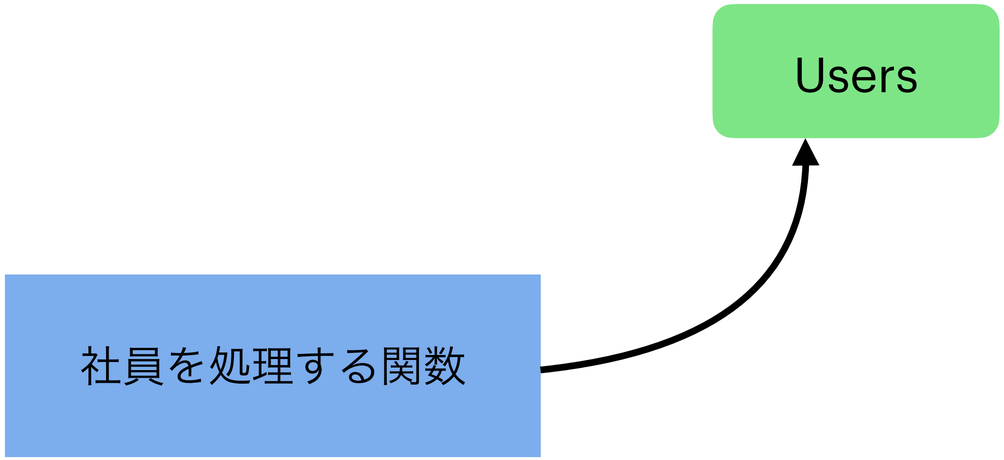
ドメインを表す名前空間に関数定義すればよくない？

Clojureでは、関数は「名前空間」で区分けすることができます



何がドメインかは、関数が決める

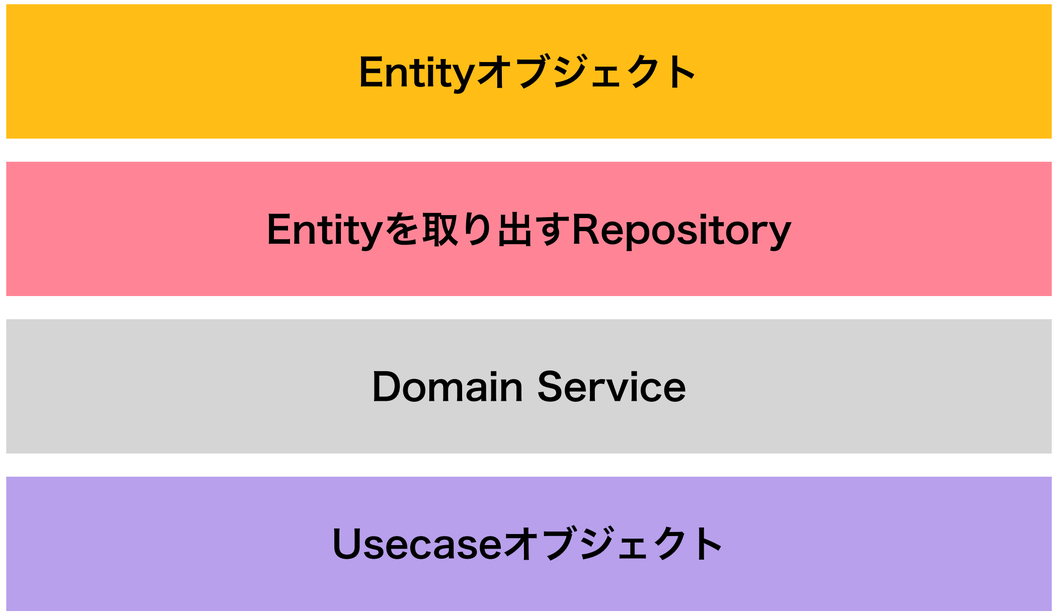
関数がそのデータを社員として扱うなら、そのデータは社員



データが社員クラスだから社員なのではない

データそれだけでは、ただのデータで、何者でもない

これらは、オブジェクト指向のデザインパターンだった



オブジェクト指向言語の特性や制約が生んだものであって、システム設計上必須のものではなかった

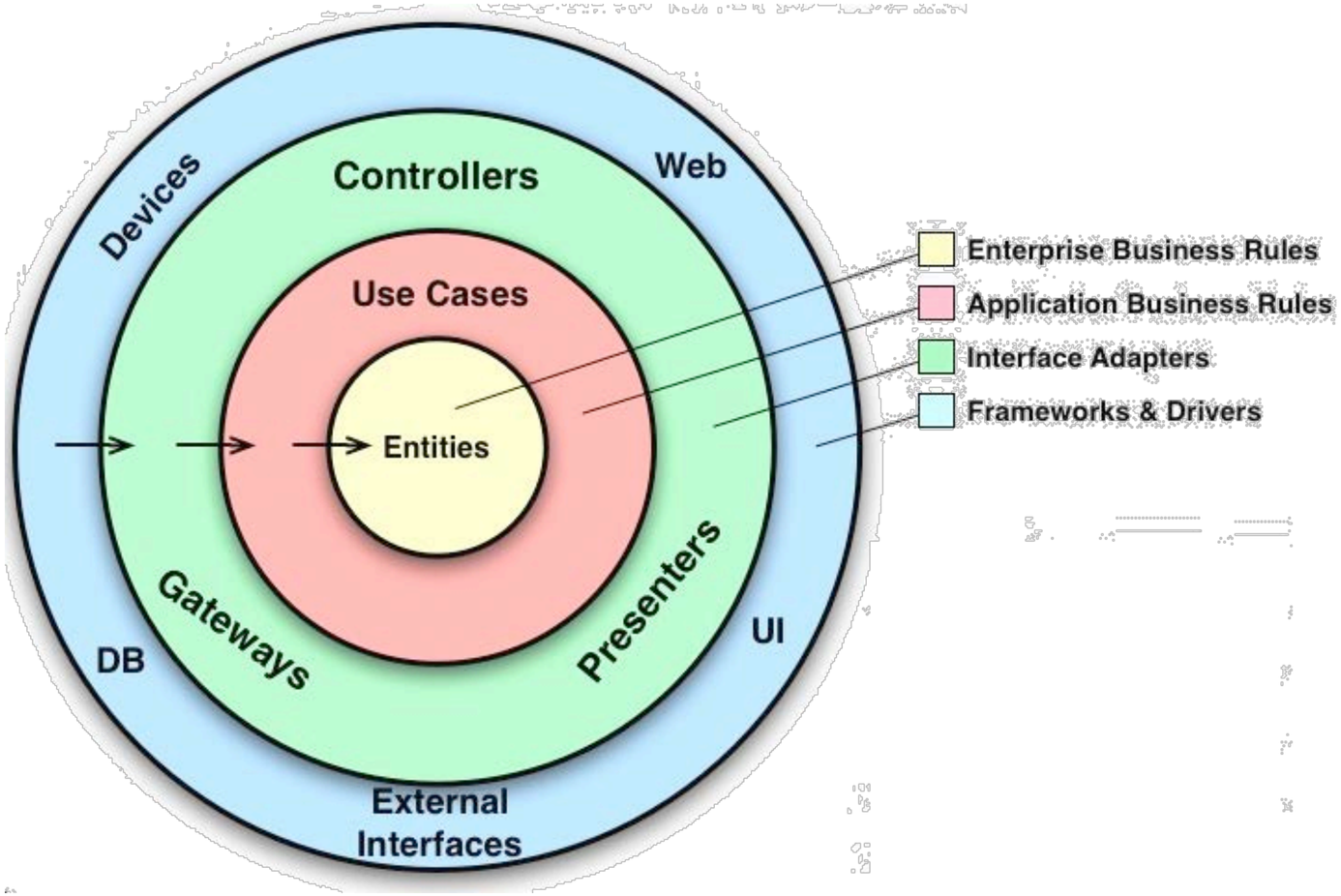
デザインパターンは、条件が変わると消滅することがある

でも、それでも消えなかったものも
ありますよね

その部分が、言語を超えて通用するアーキテクチャ

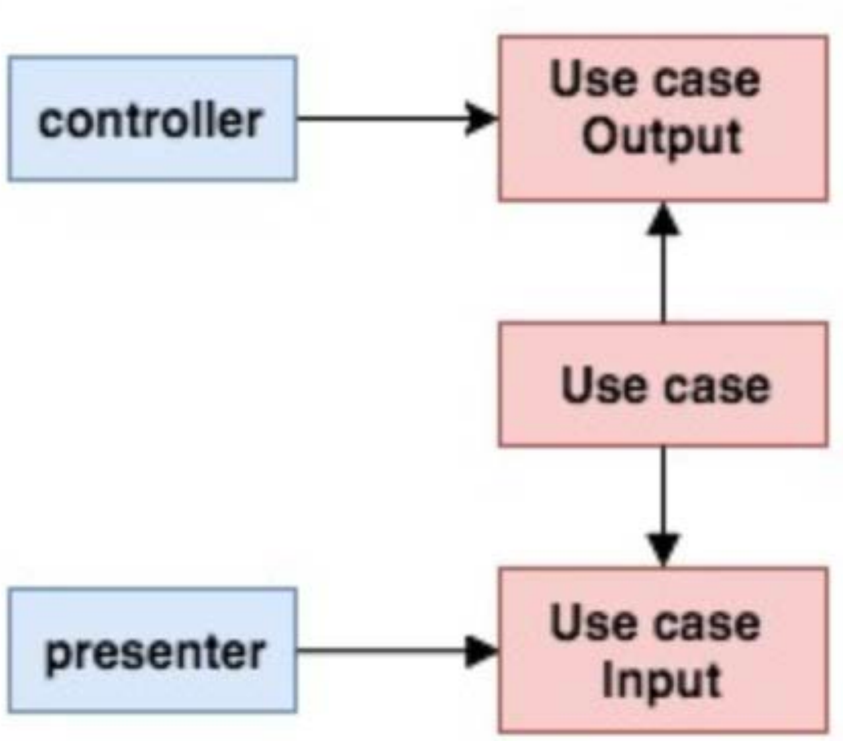
“Clean Architecture”というアーキテクチャはない

あの図は例の一つです



この図の通りやってる人どのくらいいますか？

特にWebアプリケーション開発してる人



割と都合悪いところは無視してません？

でもそれはそれでも構わない場合もある

SOLID原則

2000年ごろにUSENETで誕生（名前がついたのは2004年）

- S - Single-responsiblity Principle 単一責任の原則
- O - Open-closed Principle オープン・クローズドの原則
- L - Liskov Substitution Principle リスコフの置換原則
- I - Interface Segregation Principle インターフェイス分離の原則
- D - Dependency Inversion Principle 依存関係逆転の原則

“SOLID原則は、関数やデータ構造をどのようにクラスに組み込むのか、そしてクラスの相互接続をどのようにするのかといったことを教えてくれる。「クラス」という用語を使ったからといって、これらの原則がオブジェクト指向ソフトウェアにしか通用しないわけではない。ここでいうクラスとは、単にいくつかの機能やデータをとりまとめたものを指しているにすぎない。「クラス」と呼ぶかどうかは別として、どのようなソフトウェアシステムにもそのような仕組みはあるはずだ。SOLID原則は、そうした仕組みに適用するものである”

Ｒｏｂｅｒｔ Ｃ．Ｍａｒｔｉｎ,角 征典,高木 正弘. Clean Architecture　達人に学ぶソフトウェアの構造と設計 (Japanese Edition)

単一責任の原則

個々のモジュールを変更する理由がたったひとつだけになるように、ソフトウェアシステムの構造がそれを使う組織の社会的構造に大きな影響を受けるようにする。

オープン・クローズドの原則

ソフトウェアを変更しやすくするために、既存のコードの変更よりも新しいコードの追加によって、システムの振る舞いを変更できるように設計すべきである

リスコフの置換原則

要するに、交換可能なパーツを使ってソフトウェアシステムを構築するなら、個々のパーツが交換可能となるような契約に従わなければいけないということ。

インターフェイス分離の原則

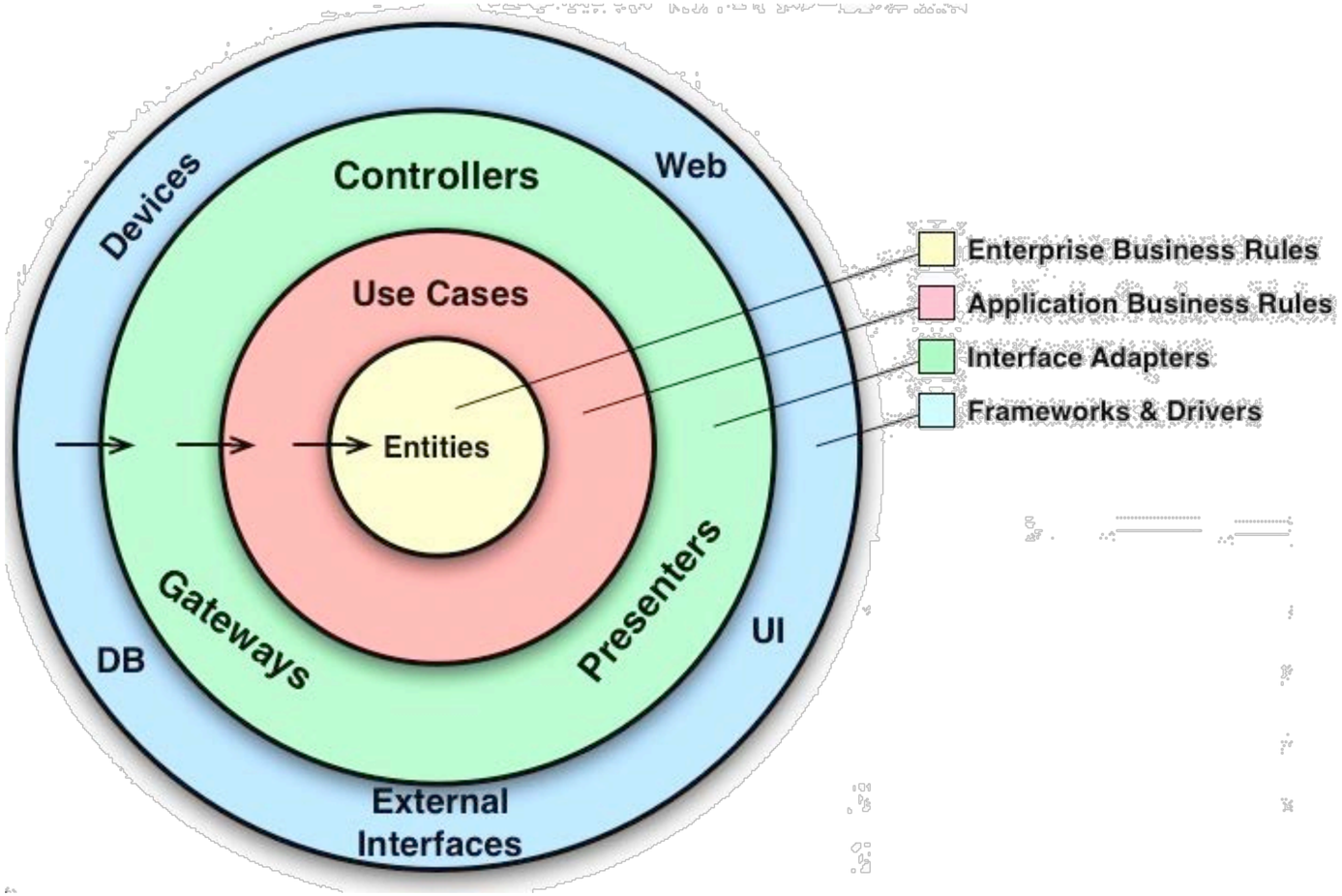
ソフトウェアを設計する際には、使っていないものへの依存を回避すべきだという原則。

依存関係逆転の原則

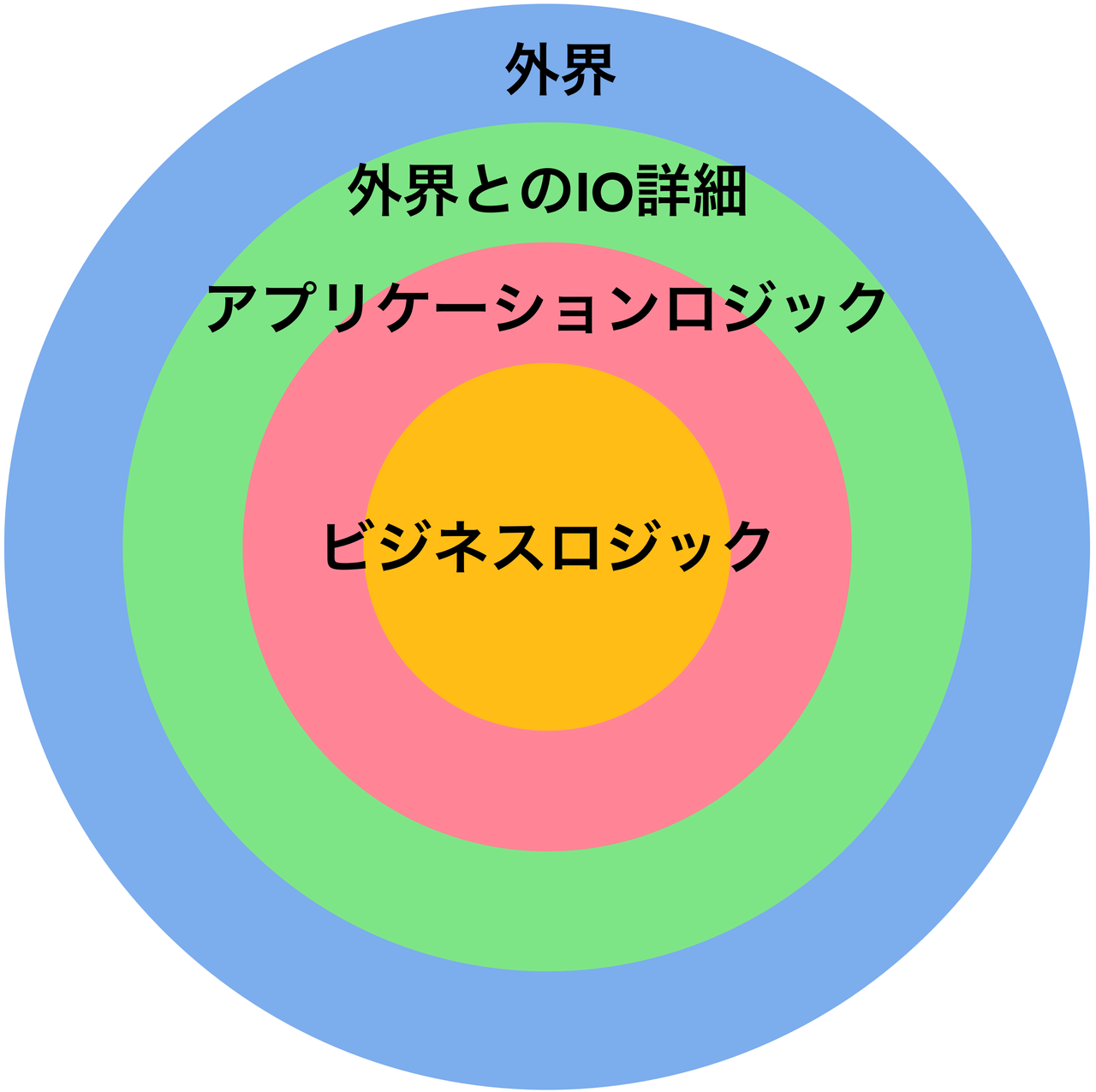
上位レベルの方針の実装コードは、下位レベルの詳細の実装コードに依存すべきではなく、逆に詳細側が方針に依存すべきであるという原則。

SOLID原則をシステムに当てはまるとこうなるよね

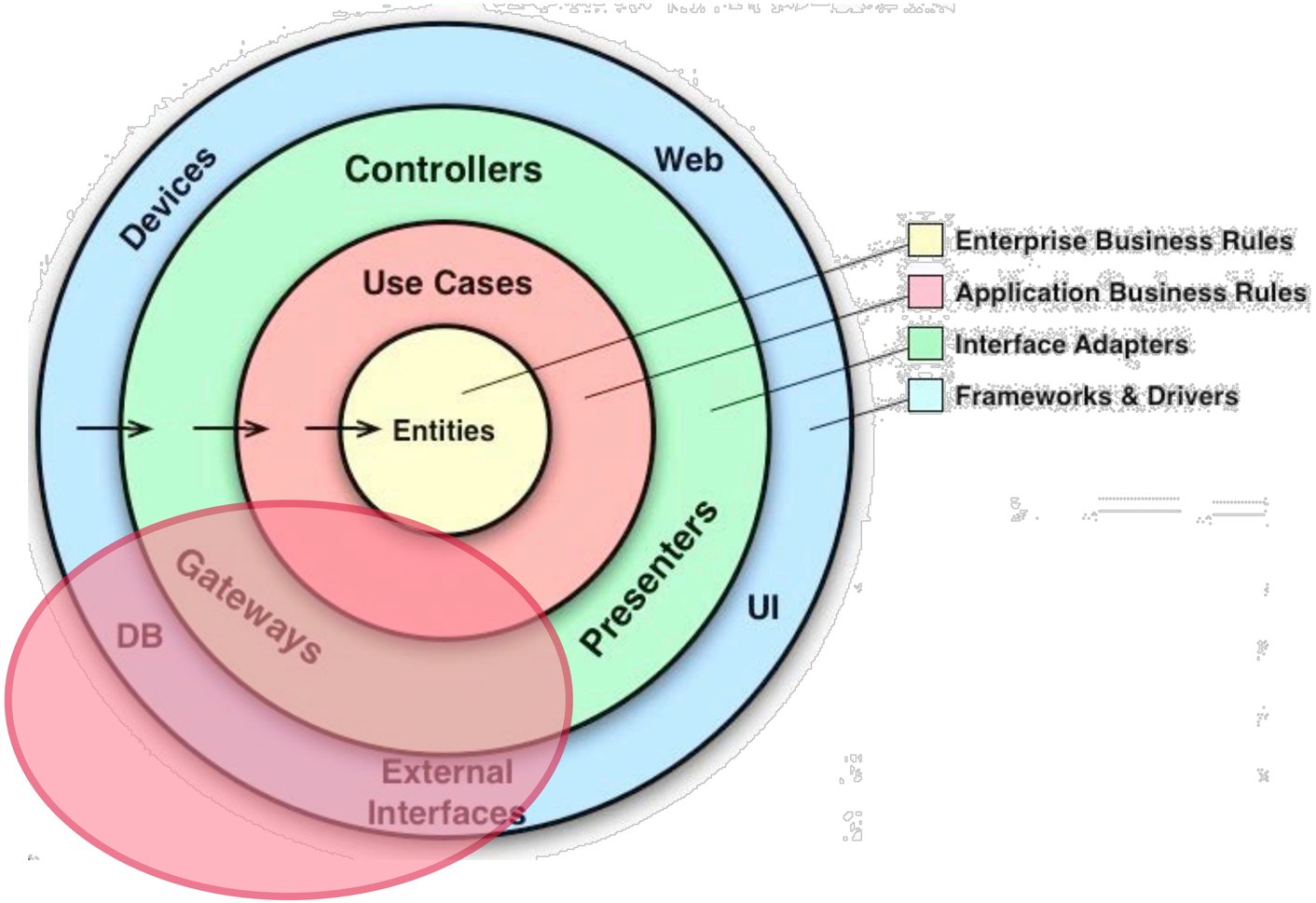
と言うのがこの図



ぶっちゃけ色分け以外の文字はどうでもいい...


外界
外界とのIO詳細
アプリケーションロジック
ビジネスロジック

でも一番複雑なプログラム、ここにあったりしませんか？



一番複雑な部分を、**Gateway**の向こう側に吹っ飛ばしてしまってるとも言える

それじゃ、システムの複雑さは改善されないですよね？？

この図の通りにクラス分けできているか？

よりも

自分のプログラムは

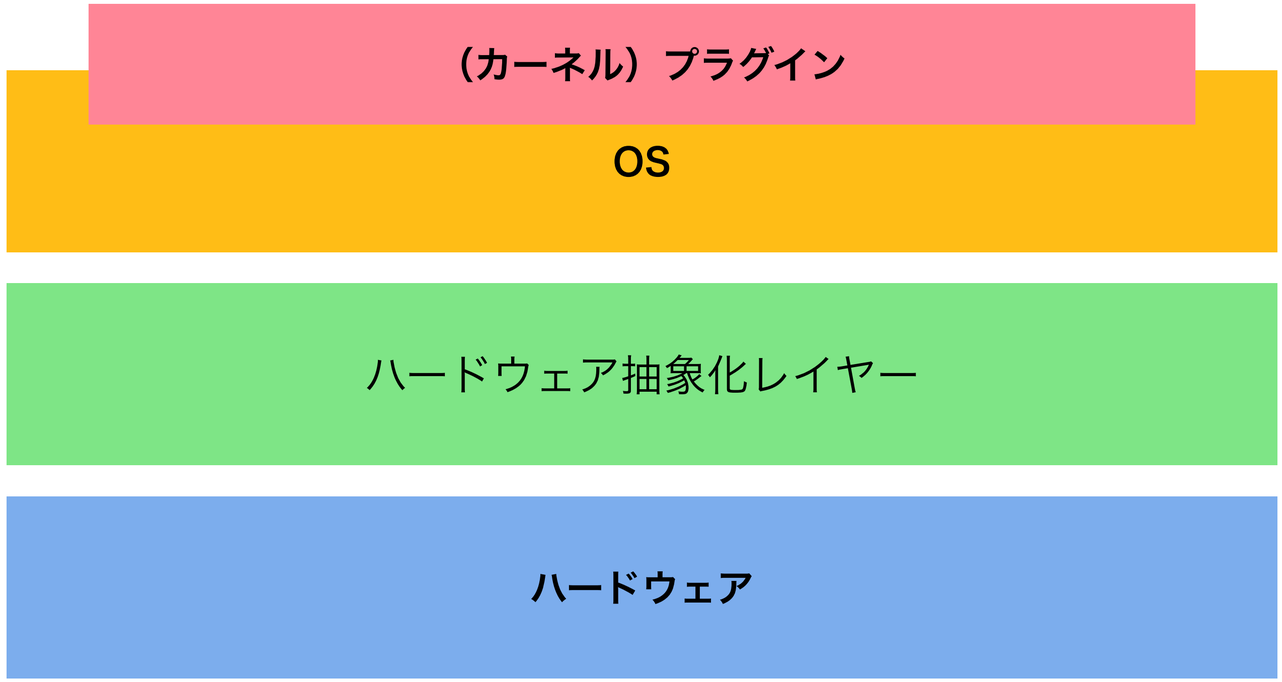
SOLID原則を守れているか？

の方が大事。

- 下位レイヤーとロジックを分離できているか？
  （**インターフェイス分離の原則**）
- ロジックが下位レイヤーの詳細に依存するのではなく、下位レイヤーの方がロジック側の提供する仕様に合わせているか？
  （**依存関係逆転の原則**）
- 下位レイヤーの変更を行わずに、コードを追加することで、機能の拡張ができるか？
  （**オープン・クローズドの原則**）

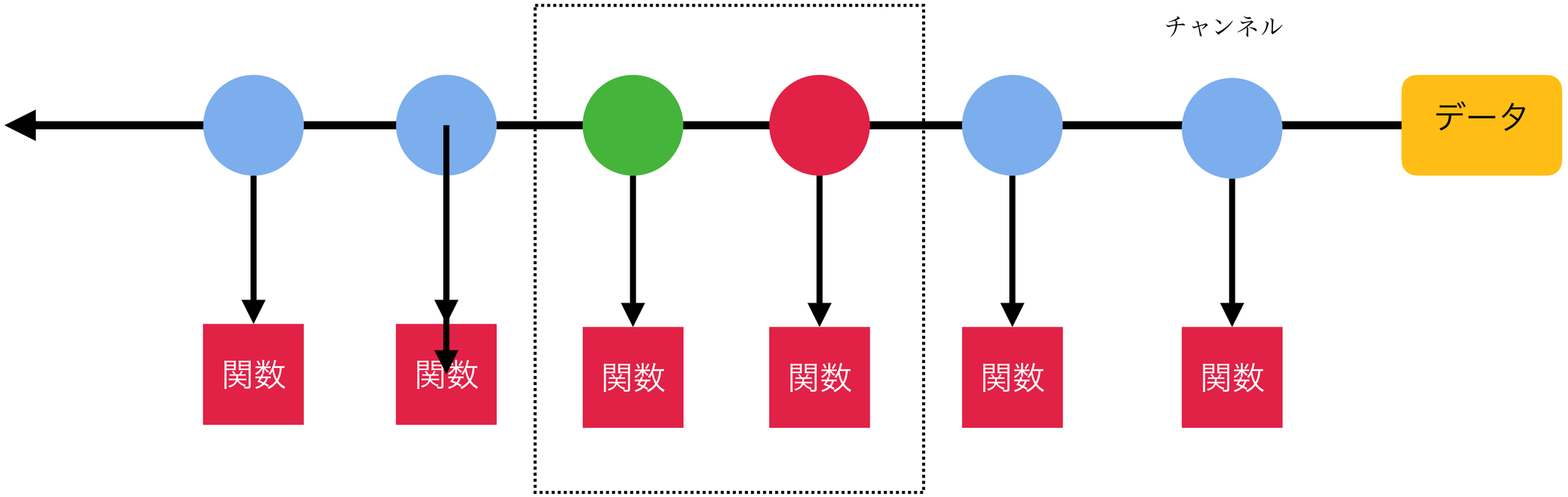
こういった理屈は、言語が変わっても
そのまま適用できる

OSにだって適用できる



Webアプリケーション以外にだって適用できる

Clojureのチャンネル・パイプラインの実装



既存のチャンネル＋関数を変更することなく、
チャンネルを追加することで、機能を拡張・改変できる

もちろんGatewayの向こう側に
吹っ飛ばした複雑性にも
適用できる

言語を超えて通用するものを見つけるために

パラダイムの違う言語をやろう

今オブジェクト指向言語ばっかり使っているなら
オブジェクト指向言語でないものをやろう

今回言いたかったこと

言語を超えても有効なものこそが、
システムの基盤となるアーキテクチャ理論になる

モデリングは楽しいけども、
その部分を無視したモデリングは多分何も解決してくれない

というわけで

とりあえず

Clojure

やってみませんか？

楽しいよ...

終わり