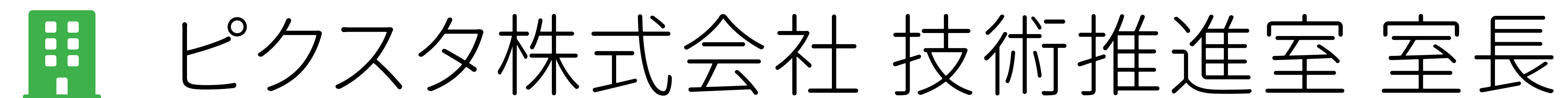
Ruby on Railsの正体と向き合い方

Yuichi Goto (@_yasaichi)

March 23, 2019 @ Rails Developers Meetup 2019

self.inspect

他社さんにおける技術基盤の
ようなチームです



Special thanks to @t_wada! 🙇

名無し募集中。。。：2011/12/06(火) 18:32:24.37 0

テスト書いてないとかお前それ
@t_wadaの前でも
同じこと言えんの?

本発表は、ピクスタの技術顧問である @t_wada（和田卓人）さんとの議論を通じて考えてきたことをまとめたものです。きっかけを頂き感謝しています。

Agenda

1. **背景と目的** 👈
2. 第一部: Ruby on Railsの正体
3. 第二部: Ruby on Railsとの向き合い方
4. まとめ

Rails Developers Meetup（Railsdm）と私

2017/05 Railsdm #1開催

2017/12 Railsdm 2017開催、LT枠で初登壇

2018 Railsdm 2018 Day 1~4開催

一参加者として楽しむ

2019/03 Railsdm 2019開催、2回目の登壇

過去のRailsdmにおけるRailsとの「向き合い方」に関する発表（一部）

レールの伸ばし方

@willnet

「Railsでまだ消耗しているの？」
—僕らがRailsで戦い続ける理由—

2017.12.09 Rails Developers Meetup 2017

Rails Developers Meetup 2017　1

Microservices Maturity Model on Rails

森 久太郎 (@qsona) 株式会社FiNC

2018/03/24 Rails Developers Meetup 2018: Day1

Realworld Domain Model on Rails

@joker1007

フォームオブジェクトとの
向き合い方

2018-07-11
Rails Developers Meetup 2018 Day 3 Extreme

諸橋恭介 @moro

1

ApplicationModel
のある風景

Hiroyasu Shimoyama

Web engineer at giftee inc.
Twitter@_h_s_

※ 各画像の出典は本発表資料の末尾に記載

過去発表の振り返りと所感

- 前提: Railsは小〜中規模の開発に向いているが、どこかで限界が来る
- 焦点: この限界に対して、コード・アーキテクチャレベルでどう対処するか
- 所感:
  - 具体的な対処法(=向き合い方)に関しては多くの議論がされてきた
  - **「いつ、なぜ限界を迎えるのか?」という前提に関する議論はまだ少ない**

本発表の目的とアプローチ

- 目的:
  - **「Railsはいつ、なぜ限界を迎えるのか?」を明らかにする**（「正体」編）
  - 明らかにした内容をもとに、**既存の対処法を俯瞰的な視点から捉えて掲示する**ことで、未来の有益な議論につなげる（「向き合い方」編）
- アプローチ: DHH氏の過去のインタビューや著作の発言を起点とした考察

Agenda

1. 背景と目的
2. **第一部: Ruby on Railsの正体 👈**
3. 第二部: Ruby on Railsとの向き合い方
4. まとめ

正体に迫る3つのキーワード

① Basecamp	Railsに働く大きな力学
② 早くてキレイ	DHHがRailsで目指したもの
③ 密結合	DHHがRailsで採ったアプローチ

キーワード①
Basecamp

Basecampとは

- アメリカのシカゴに本社を置くBasecamp社（旧名37signals、DHHは同社のCTO）の提供するプロジェクト管理ツール [1]
- 2003年、当時本業だったWebデザインのコンサルティング業務で起きていたコミュニケーションの問題を解決するために開発した [2]
- 当初は社内と既存顧客の間で利用していたが、「（顧客の）社内でも使いたい」という声を受け、2004年にサービスとして提供を始めた [2]

DHHの関わり

“**Basecamp（注1）という新しいプロダクトの開発のとき，自分が開発環境を決められるようになり，**それなら一番美しいソースコードを書ける言語にしようということでRubyにしたんです。”

出典: 小飼弾のアルファギークに逢いたい♥ #2（2006）

“I did **all the original programming for Base Camp**”

出典: Millionaire Story: David Heinemeier Hansson（2011）

※ 太字強調は引用者によるもの

Railsとの関係

“「Basecamp」と呼ぶ，Webベースのプロジェクト管理ツールを開発しました。**Ruby on Railsはそのために開発したフレームワークで，最初は公開するつもりはなく，内部だけで使用していました。**”

出典:「美しいコードを書けるからRubyを選んだ」
---Ruby on Rails作者David Heinemeier Hansson氏（2006）

“Railsを作ったのは、**自分の仕事をより良く、より速くするためです。**”

出典: Ruby on Rails: DHHのインタビュー（2005）

※ 太字強調は引用者によるもの

これらから言えること

- DHHはBasecampと後にRailsとなるフレームワークを一人で作っていた
- DHHはBasecampをより良く、より速く実装するためにRailsを作った

当時のBasecampが置かれていた、**「少人数のスタートアップでのプロダクト開発」という状況がRailsの設計に影響している**

キーワード②

早くてキレイ

DHHがRailsを作るまで

1. Railsを作る前は、DHHは主にPHPとJavaを書いていた [3]
2. Railsも最初PHPを使って作っていた(!)が、不満を感じていた [4]
3. Martin FowlerやDave Thomasの書いた記事に影響されて、試しにRubyを使ってみることを決める [5]
4. 1週間ほどですっかりRubyにハマり、後のRailsを作り始める [5]

DHHの問題意識

"**私はこの2つのソフトウェア開発手法に挟まれてたんですね。**PHPに代表されるような「早いけど汚い」手法と、Javaに代表されるような「遅いけどキレイ」な手法にです。それで、**両者を組み合わせたら、究極の目標である「早くてキレイ」になるんじゃないかと思ったわけですよ。**まあ、せめて、両方の人たちにアピールするくらいはできるんじゃないかなって。"

出典: Ruby on Rails: DHHのインタビュー（2005）

※ 太字強調は引用者によるもの

DHHがRailsで目指したもの

“I was mostly doing PHP on my own, and I worked at a Java shop for a period of time. It was J2EE to some extent and otherwise Java in general. Those were the two forming influences. **With Ruby On Rails, I tried to form the best of both worlds to make it as quick as PHP and as solid and clean as something like Java.**”

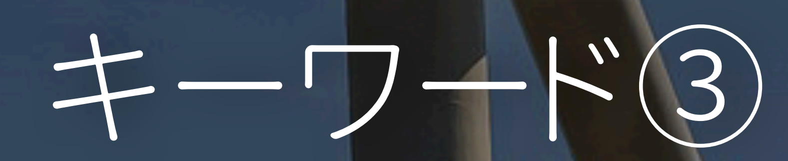
出典: Rails creator on Java and other 'junk'（2007）

※ 太字強調は引用者によるもの

「早くてキレイ」は本当に実現できるのか?

- ソフトウェアの開発速度と品質は(ある程度)トレードオフの関係にある
- 原理的に解けない問題を解こうとする場合、何らかの仮定や制約を置いて対処することが多い

「早くてキレイ」を実現するために、**DHHも何かを妥協しているはず**



キーワード③

密結合

Railsがある程度妥協しているもの

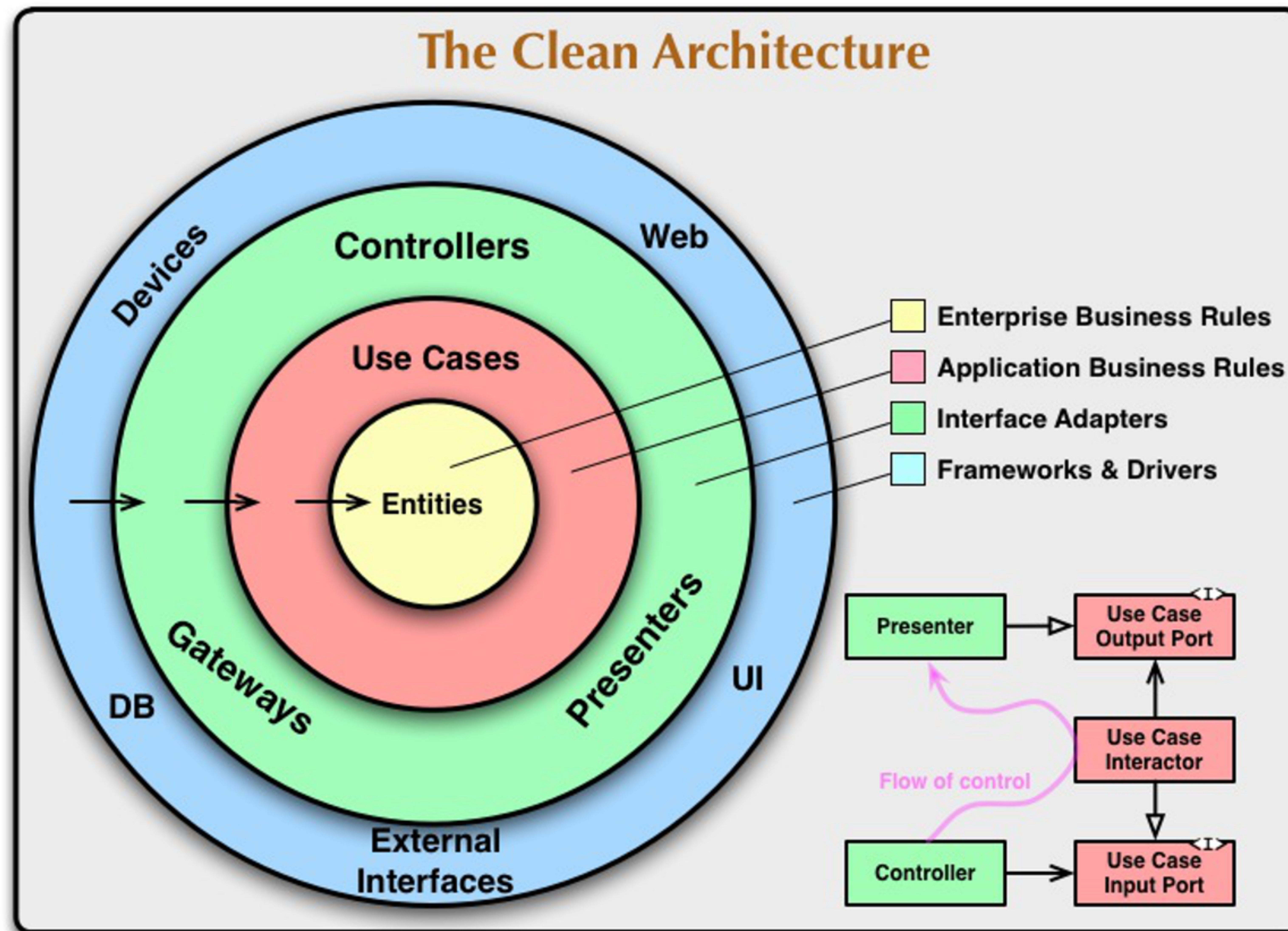
1. **柔軟さ**
   - 例: CoC（Convention Over Configuration、「設定より規約」）
   - 柔軟さと引き換えに考える・書く量を減らしたことは、何度も話されてきた
2. **疎結合（↔密結合）であること**
   - 「Railsは密結合な設計だ」と言われても、ピンと来ない方もいるのでは

ずっとRailsだけで仕事してきた方とか

密結合度合いを比較で理解する

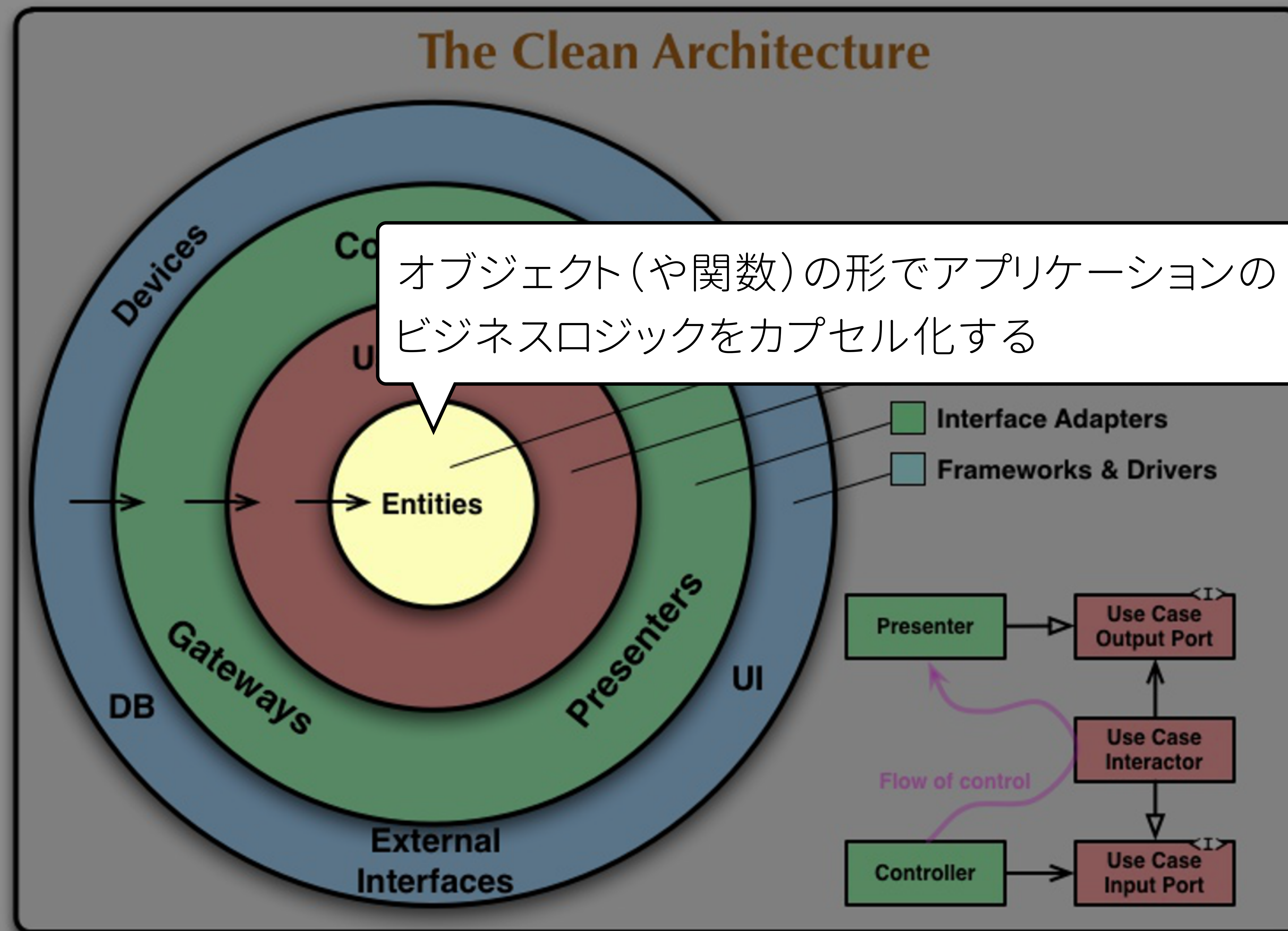
- **何らかの機能をRailsとHanamiでそれぞれ実装し、比較してみる**
- なぜHanami?
  - Hanamiはクリーンアーキテクチャをほぼそのまま実装している [6]
  - クリーンアーキテクチャは疎結合な設計の代表例のひとつである
  - どちらもRuby向けのWebフレームワークのため、**設計の差が際立つ**

[脇道] そもそも、クリーンアーキテクチャとは?🤔



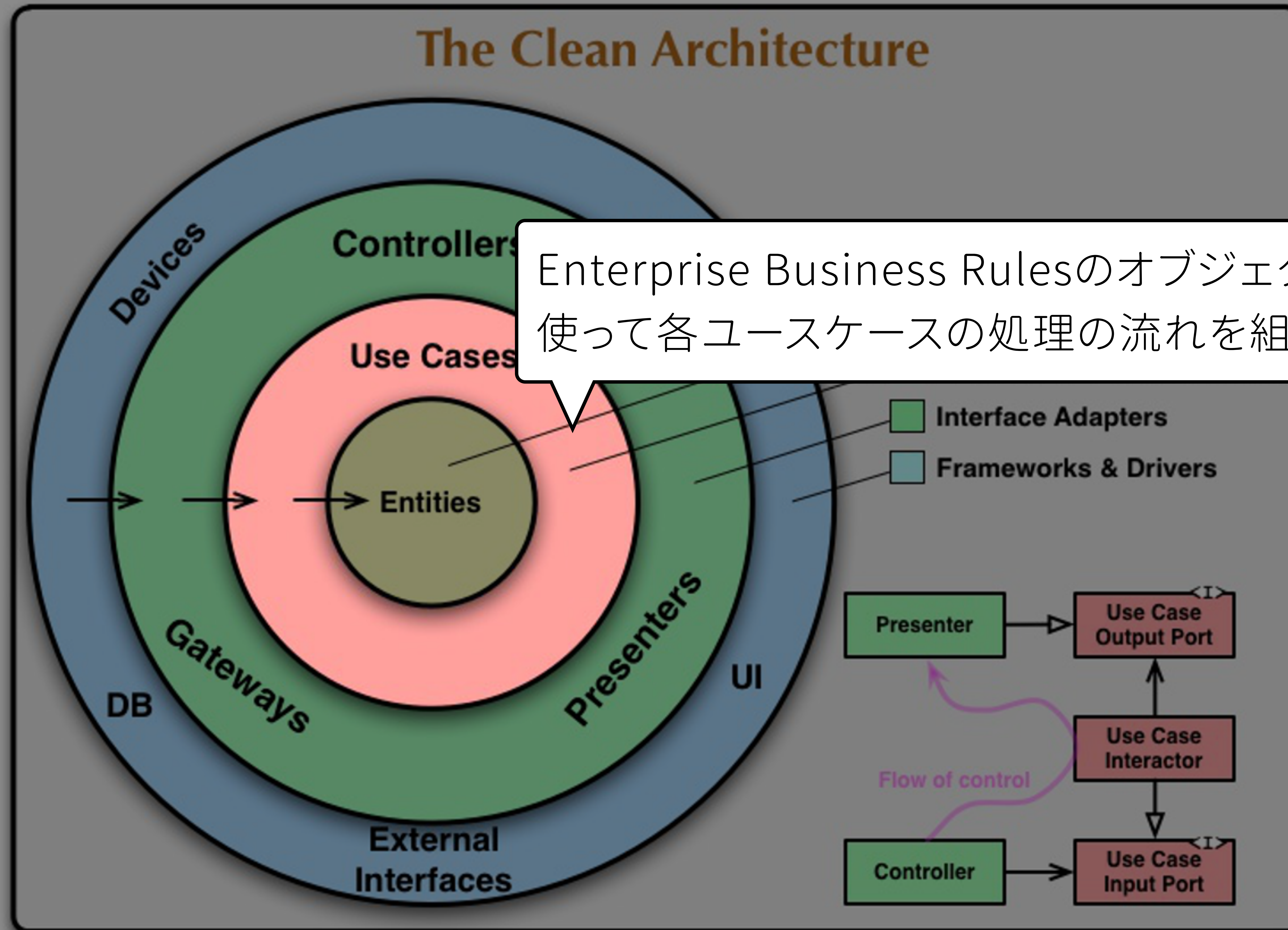
画像出典: Robert C. Martin "The Clean Architecture" (2012) [7]

Enterprise Business Rules（例: Entity, Value Object）



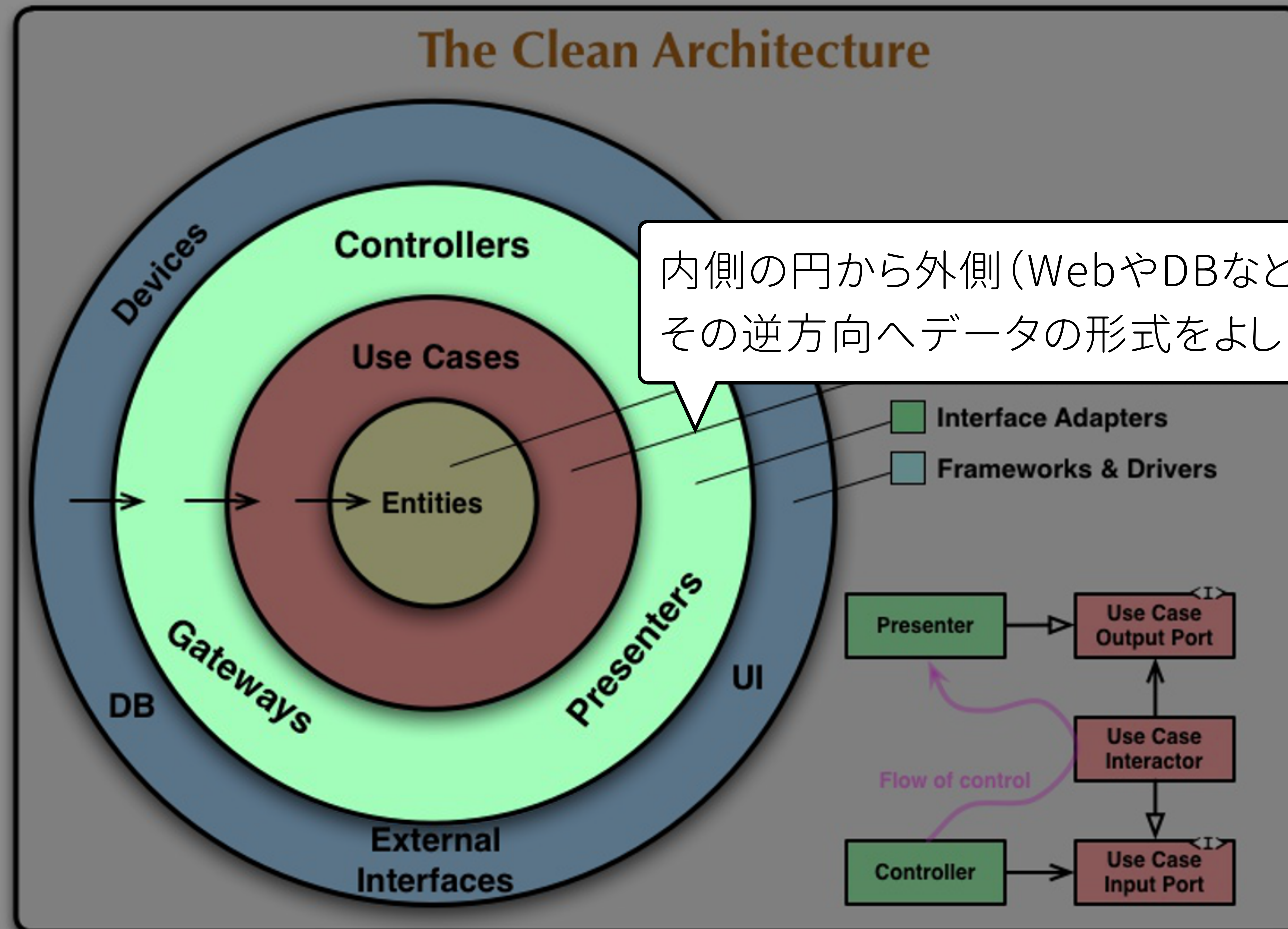
オブジェクト（や関数）の形でアプリケーションの
ビジネスロジックをカプセル化する

Application Business Rules（例: Interactor）



Enterprise Business Rulesのオブジェクト群を
使って各ユースケースの処理の流れを組み立てる

Interface Adapters（例: Controller, Presenter, Repository）



内側の円から外側（WebやDBなど）の円、または
その逆方向へデータの形式をよしなに変換する

今までの説明で完全に理解できた
はずなので、本題に戻ります👮‍♂️

お題: 簡易ユーザー登録機能

正常系のフローは次の通り:

1. ユーザーはメールアドレスを入力して送信ボタンを押す
2. 送信されたメールアドレスの形式が正しいことを確認する
3. アドレス存在確認用のトークンを作り、ユーザーにメールを送信する
4. ユーザーをメールアドレス登録完了ページへリダイレクトする

https://github.com/yasaichi-sandbox/user_registration

Railsでの実装例: Controller

```ruby
class UserRegistrationsController < ApplicationController
  def create
    @user_registration = UserRegistration.new(user_registration_params)

    if @user_registration.save
      redirect_to complete_user_registrations_url
    else
      render :new
    end
  end

  private

  def user_registration_params
    params.require(:user_registration).permit(:email)
  end
end
```

Railsでの実装例: Model

```ruby
class UserRegistration < ApplicationRecord
  validates :email, format: { with: URI::MailTo::EMAIL_REGEXP }

  before_create :set_confirmation_token
  after_create :send_confirmation_instructions

  private

  def set_confirmation_token
    self.confirmation_token = SecureRandom.uuid
  end

  def send_confirmation_instructions
    UserRegistrationMailer.with(user_registration: self)
                          .confirmation_instructions.deliver_now
  end
end
```

このへんは説明のためのコードなので、真似しないように

Hanamiでの実装例: Controller

```ruby
module Web::Controllers::UserRegistrations
  class Create
    include Web::Action

    expose :error_messages

    def initialize(interactor: StartUserRegistration.new(
      mailer: Mailers::UserRegistrationConfirmation,
      repository: UserRegistrationRepository.new,
      token_generator: Utils::UrlSafeTokenGenerator
    ))
      @interactor = interactor
    end

    def call(params)
      result = @interactor.call(params[:user_registration])

      if result.successful?
        redirect_to routes.complete_user_registrations_url
      else
        @error_messages = result.errors
        self.status = 422
      end
    end
  end
end
```

メソッド呼び出しの対象がModelからInteractorへ

Hanamiでの実装例: Interactor（入力値チェック）

```ruby
class StartUserRegistration
  include Hanami::Interactor

  class Validator
    include Hanami::Validations

    validations do
      required(:email).filled(:str?, format?: URI::MailTo::EMAIL_REGEXP)
    end
  end

  private

  def valid?(params)
    Validator.new(params).validate.yield_self do |result|
      result.messages.each_key { |key| error("#{key.capitalize} is 
invalid") }
      result.success?
    end
  end
end
```

#valid?がtrueを返すと#call（後述）が実行される

Hanamiでの実装例: Interactor（処理本体）

```
class StartUserRegistration
  include Hanami::Interactor

  def initialize(mailer:, repository:, token_generator:)
    @mailer = mailer
    @repository = repository
    @token_generator = token_generator
  end

  def call(params)
    @repository.transaction do
      user_registration = @repository.create(
        email: params[:email],
        confirmation_token: @token_generator.call
      )

      @mailer.deliver(user_registration: user_registration)
    end
  end
end
```

#before_create → #save → #after_createに相当

Rails Model ≒ Hanami Interactor

- お題のようなレコード作成+α時には、**ModelはInteractorに相当する**
- Interactorの役割は、各ユースケースの処理の流れを組み立てること

Railsの**Modelは特定のユースケースと密結合している**

Interactor化を支える3つの技術

- **Active Record（AR）パターンの適用**: Modelとテーブルを1:1対応させ、ビジネスロジックの記述も許すことで、EntityとRepositoryの役割を実現
- **AR Validationsの発明**: Modelの各属性が満たすべき条件を書く形で、Interactorでの入力値チェック相当の処理を実現
- **AR Callbacksの発明**: ModelのCRUD操作に対するフックの形で、Interactorでのユースケースの処理組み立て相当の処理を実現

C(R)UD操作とコールバックが自動で同一トランザクションになるのがキモ

Clean ArchitectureとRails Modelの対応

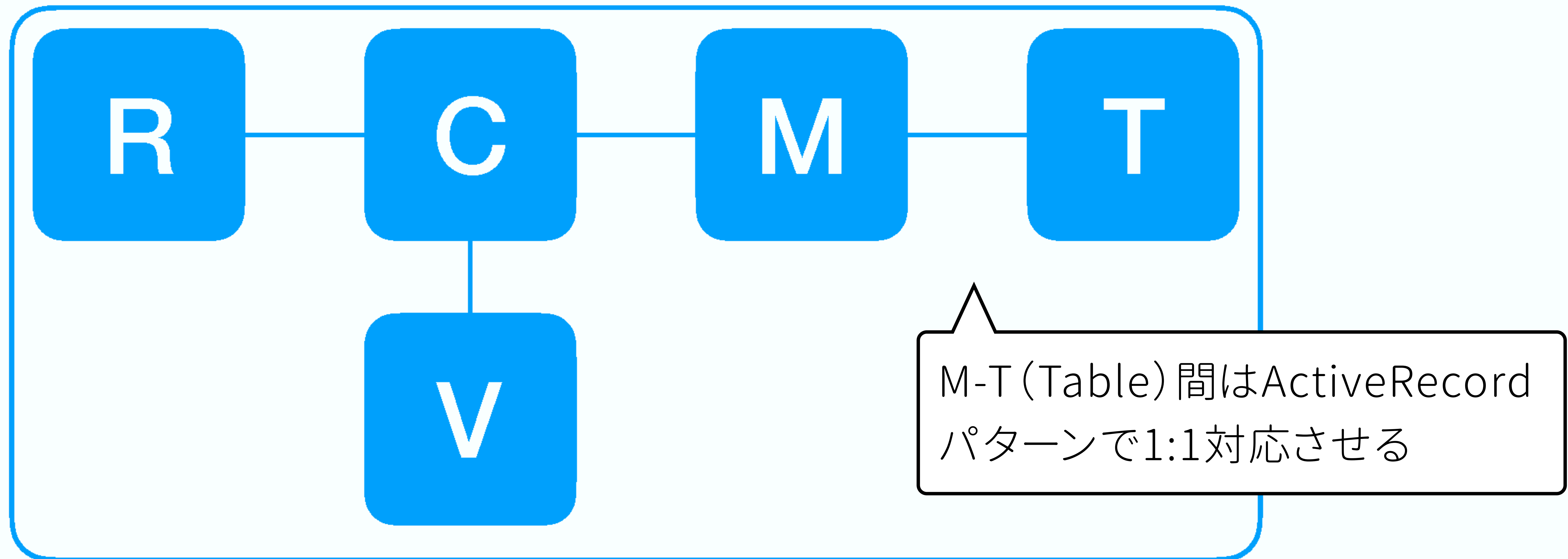
	Clean Architecture	Ruby on Rails
ビジネスロジックの記述	Entity	Model
ユースケースの組み立て	Interactor	Model (AR Callbacks)
入力値のバリデーション	Interactor	Model (AR Validations)
DBアクセス・データ変換	Repository	Model (AR Query Interface)

ダメ押しの一手: RESTfulルーティング

- ActiveRecordとそのValidations/Callbacksはv1.0時点で既に存在した
- v1.2でRESTfulルーティング(resources)が導入され [8]、URLで表されるリソースとModelが1:1対応した

RESTfulのルーティングの導入で全てのコードがModelのCRUD操作を中心に整理されたことで、現在のRailsの原型が完成した

全てのコードがModelへのCRUD操作を中心に整理された状態



Ruby on Railsの正体

- 次の2つの方法を組み合わせて、ModelのCRUD操作を中心にコードを整理して考える・書く量を減らすことで、「早くてキレイ」を実現した
  - RESTfulルーティングとActiveRecordパターンによって、**URLで表されるリソースからDB上のテーブルまでが密結合する構造を作った**
  - ActiveRecordパターンとそのValidations/Callbacksによって、**ビジネスロジックとその組み立て処理を全てModelに書けるようにした**

「密結合は悪」なのでは?

- 大規模なアプリケーションの分業開発においては間違いなく避けるべき
- 一方、DHHがRailsを作ったときに置かれていた状況は「少人数のスタートアップでのプロダクト開発」

密結合にしても問題になりにくい状況だったので、これと引き換えに**スタートアップでは最も重要な開発速度を出せるような設計にした**

師曰く: Ruby on Railsの功績

“少なくともスタートアップ企業にとってスピードは本当にクリティカルな力なので、**もし密結合の状態でも速く走れるソフトウェアの構造があるのであれば、それはゆっくり安定して継続的に歩いていく疎結合のソフトウェア設計より強いということをRailsはある程度証明していたわ**けですね。そしていま、その構造のまま大きくなるとすごく大変になるということも証明している。”

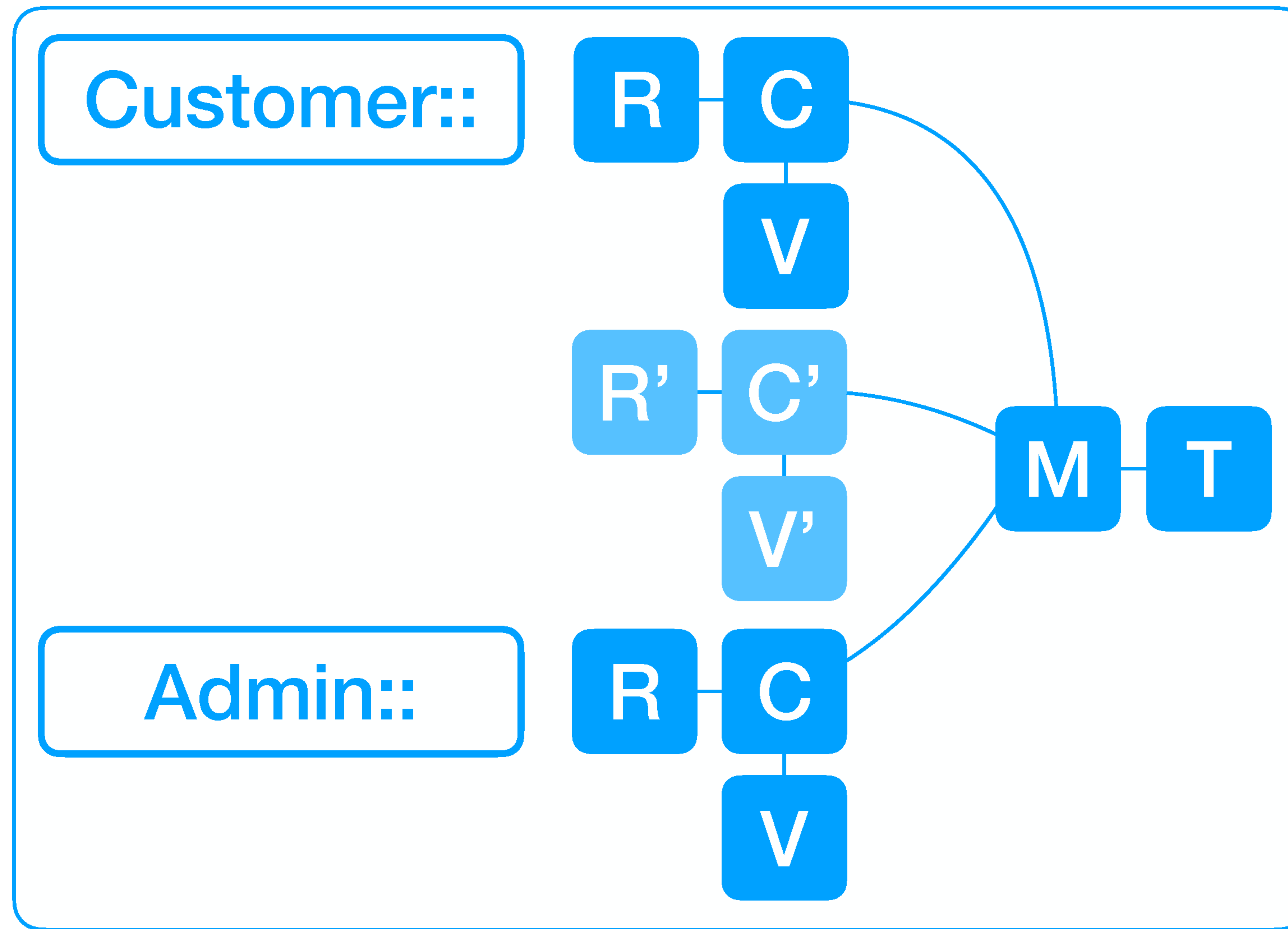
出典: マニアが潰したテスト駆動開発～『健全なビジネスの継続的成長のためには健全なコードが必要だ』対談（5）(2018)

※ 太字強調は原文のもの

Railsはいつ、なぜ限界を迎えるのか?

- いつ: あるModelが複数の異なるユースケースでC(R)UD操作されるようになったとき
- なぜ: あるModelに書かれたValidations/Callbacksは特定のユースケースと密結合しているため
- 何が辛いのか: あるユースケースに特化したModelの中で、別のユースケースの事情を考慮したコードを書かなければいけないこと

理想状態から離れ、限界が見え始めてくる状態



画像出典: ApplicationModel のある風景 (2018) [9]

限界の表出の仕方

1. 特定の条件でValidations/Callbacksを実行する or スキップする
   - 例: `if: :condition?`, `on: :context`, `save(validate: false)`
   - Modelに複数のユースケースの事情が詰め込まれていることを示す
2. Controller内に`ApplicationRecord.transaction`を書く
   - ModelのCRUD操作を中心に処理が組み立てられなくなったことを示す

第一部完

[PR] ピクスタはクリエイティブプラットフォームを創る会社です

PIXTA ピクスタ株式会社

企業情報 事業紹介 IR情報 ニュース 採用情報 ピクスタ+

ホーム > 企業情報 > 事業紹介

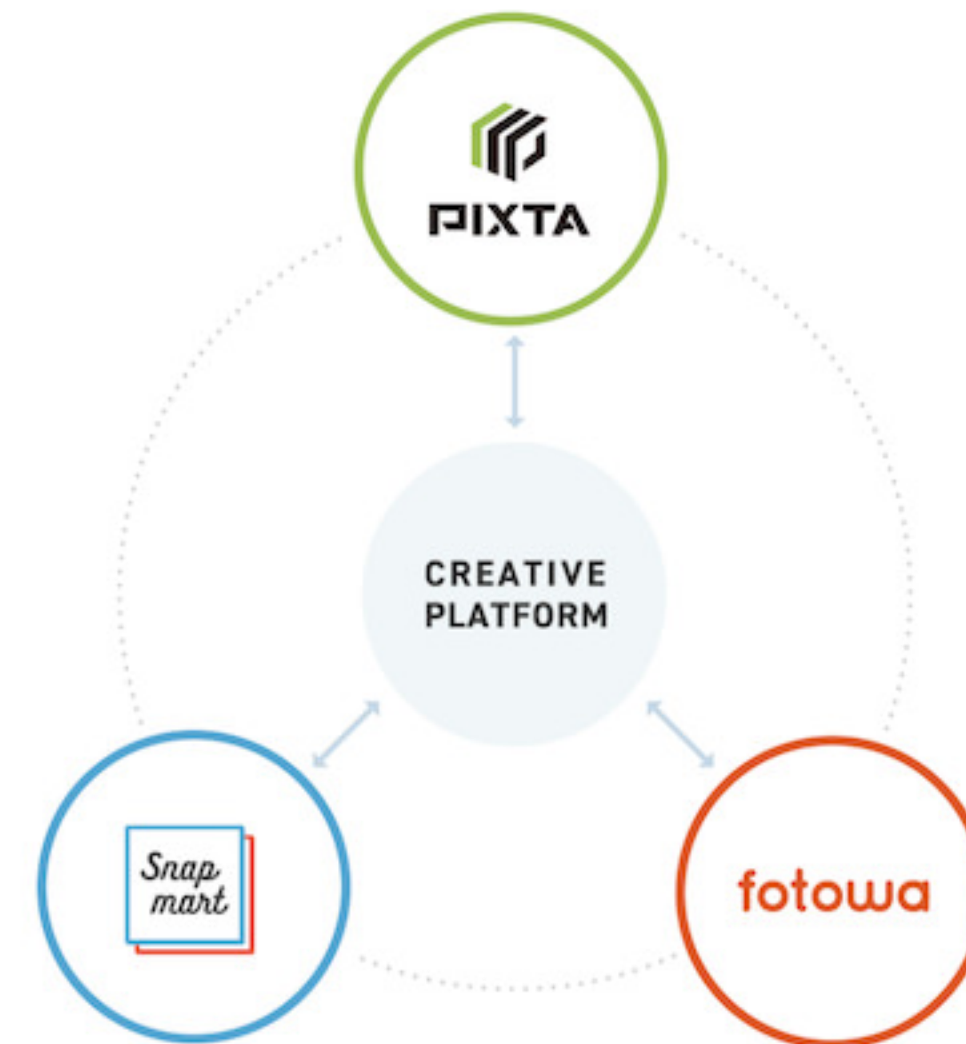
事業紹介

BUSINESS

3つ全てのプラットフォームでRailsを利用。
今年もRubyKaigiに協賛しています（6回目）

当社グループは「クリエイティブ・プラットフォーム事業」を主な事業とし、現在、PIXTA、fotowa、Snapmartという3つのクリエイティブプラットフォームを運営しています。

画像・動画・音楽の素材サイト	自然でオシャレな出張撮影なら	コンテンツ流通に"素人革命"を
PIXTA	fotowa	Snapmart



コーポレートサイトトップ: https://pixta.co.jp

[採用情報] 技術推進室のソフトウェアエンジニアを絶賛募集中

PIXTA 採用情報

代表メッセージ　働く仲間　働く環境　データで見るピクスタ　募集要項　ピクスタ＋　エントリー

採用情報 | 正社員

【PIXTA】技術基盤エンジニア

採用トップ ＞ 募集要項 ＞ 技術基盤エンジニア

仕事内容 JOB DESCRIPTION

技術基盤エンジニア

PIXTAが目指すもの

「PIXTA」は、インターネット上でクリエイターから集めた写真・イラスト・動画等のデジタル素材を、素材を必要とする法人・個人向けに販売するオンラインマーケットプレイスです。現在、25万人以上のクリエイターから集められた3,000万点以上の素材が、日本やアジアを中心とした世界各国の購入者に利用されています。

PIXTAでは、プロ・アマチュアを問わずオンラインで素材を投稿することができるため、会社員、主婦、学生、シニアなどのアマチュアクリエイターからプロのフォトグラファー、イラストレーター、ビデオグラファーまで、国内外の幅広い層のクリエイターが時間や距離、経歴や経験にとらわれることなく活躍しています。

募集中の職種一覧: https://recruit.pixta.co.jp/careers

Agenda

1. 背景と目的
2. 第一部: Ruby on Railsの正体
3. **第二部: Ruby on Railsとの向き合い方** 👈
4. まとめ

向き合い方は大きく2種類

1. **コードレベル**

   これまで説明したRailsの限界を**正面から解決しようとする**アプローチ

2. **アーキテクチャレベル**

   これまで説明したRailsの限界を**そもそも回避しようとする**アプローチ

過去に提案されたコードレベルでの向き合い方

- **ActiveRecordの分割**（by @hanachin） [10]

  責務に応じてDB上の同一テーブルを参照する複数のModelを作る

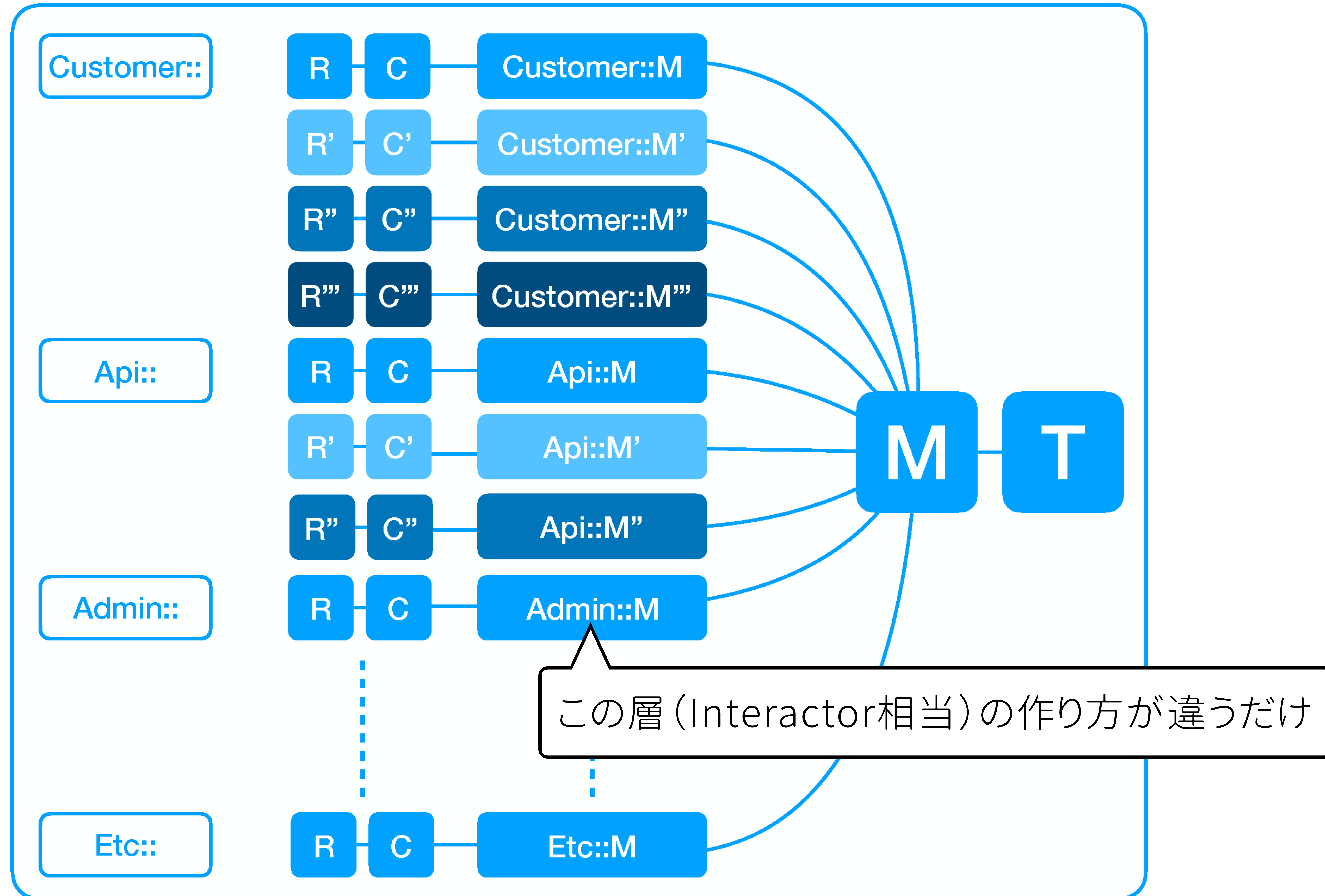
- **Application Modelの導入**（by @_h_s_） [9]

  DB上のテーブルに紐付かないController専用のModelを作る

- **Form/Service（その他様々なPORO）の導入** [11][12][13]

  Controllerからの単一メソッド呼び出しで処理一式を実行する層を作る

共通点: Controller or Actionと1:1対応する新しい層を導入する



画像出典: ApplicationModel のある風景 (2018) [9]

コードレベルでの基本方針

- ビジネスロジックとその組み立て処理を全てModelに書けるようにして、「早くてキレイ」を実現したのがRails

- Railsの辛さは複数のユースケースの事情がModelに集中することが原因

ユースケース固有の処理を書く層を作り、これらの間で共通の処理（≒ビジネスロジック）だけをModelに書くようにする

アーキテクチャレベルでの向き合い方

- 基本方針: **Railsの限界をコードレベルで解決しようとしない**
  - オプション1: アプリケーションが対象とするドメインを小さくする
  - オプション2: 限界を迎える前に複数のサブシステムへ分割する
- **組織構造とアーキテクチャは不可分なので、実行の際にはこれら両方の考慮が必要になる**ことが多く、難易度は高い

PIXTAサービスの現状と取り組み

- 2011年6月にPerlからリプレイスして現在8年目（事業自体は13年目）
- **これまで説明した限界に直面しているため、次の2つの対応を行っている**
  - 「本体」と呼ばれるリポジトリをこれ以上肥大化させないため、独立性の高い新機能は最初からサブシステムとして実装する
  - 「本体」からSoE（System of Engagement）を抽出して、巨大なSoR（System of Record）を任意の単位で分割できるようにする

事例1: 素材タイトルの多言語翻訳機能

- 翻訳タイトルを素材のタグから自動生成していたのをやめるために開発
- ドメインが小さく独立性が高いため、最初からサブシステムとして実装
- I/O boundなアプリケーションだったこともあり、**Node.jsを使って作った**

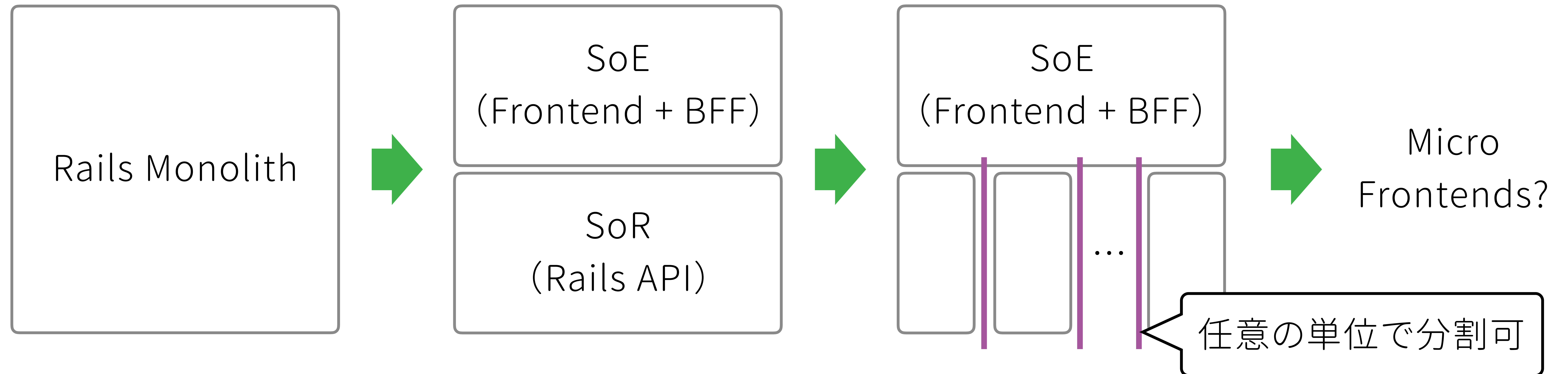
アプリケーションが対象とするドメインを小さくすると、Railsを選ぶ理由も少なくなるというジレンマがある

事例2: SoEの抽出（実施中）

- 解決したいこと: 「本体」を複数のチームで開発しているため、組織が持つ本来の開発速度を発揮できていない
- アプローチ: 「本体」を組織構造と一致するようにサブシステムに分割する
- 問題点: Railsではフロントエンドとバックエンドもまた密結合しているため、**このままの状態だとページ単位で分割せざるを得ない**

PIXTAではこの単位が組織の形とマッチしなかった（経験済み）

解決策: 先にSoEを抽出してから縦方向に分割する



最初からSoE/SoRの構成なら縦方向の分割をすぐ始められるので、
次の新規サービス開発(未定)ではBFF + Rails APIでやりたい

もうひとつの未来への道

- とはいえ、成長中のプロダクトをひとつだけ持つ一般的なスタートアップでRailsの限界が見え始めたらすぐに分割を実施することは難しそう
- 小規模での早さはそのままに、**中規模を超えてもある程度走り続けられるフレームワークをRailsの上に作る**のが現実的では？
  - Railsを構成する要素で作られている
  - **疎結合な設計に段階的に移行できる仕組みを持つ**

Laravelのサービスコンテナが参考になりそう

Agenda

1. 背景と目的
2. 第一部: Ruby on Railsの正体
3. 第二部: Ruby on Railsとの向き合い方
4. **まとめ 👈**

まとめ: Ruby on Railsの正体

- **DHHが解決したかったこと**: 少人数のスタートアップでのプロダクト開発で「早くてキレイ」を実現する
- **Railsのアプローチ**: 次の2つの方法により、ModelのCRUD操作を中心にコードを整理して考える・書く量を減らす
  - URLで表されるリソースからDB上のテーブルまでを密結合させる
  - ビジネスロジックとその組み立て処理を全てModelに書けるようにする

まとめ: Ruby on Railsの限界との向き合い方

- **Railsの限界**: あるModelが複数の異なるユースケースでC(R)UD操作されるようになると、各々の事情がそのModelに集中して辛くなる
- **限界との向き合い方**:
  - コードレベル: ユースケース固有の処理を書くための層を導入する
  - アーキテクチャレベル: 対象のドメインを小さくするか途中で分割する

参考文献

1. Basecamp, LLC "Basecamp: About our company", URL: https://basecamp.com/about
2. Basecamp, LLC "A letter from the CEO", URL: https://basecamp.com/about/story
3. IDG Communications, Inc. "Rails creator on Java and other 'junk'", URL: https://www.infoworld.com/article/2649156/rails-creator-on-java-and-other--junk-.html
4. 角征典 "Ruby on Rails: DHHのインタビュー", URL: https://kdmsnr.com/translations/interview-with-dhh/
5. "David Heinemeier Hansson interviewed by Randal Schwartz", URL: http://www.transcribed-interview.com/dhh-rails-david-heinemeier-hansson-interview-randal-schwartz-floss.html

参考文献

6. "Architecture: Overview", URL: https://guides.hanamirb.org/architecture/overview/
7. Robert C. Martin "The Clean Architecture", URL: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
8. "Rails 1.2: REST admiration, HTTP lovefest, and UTF-8 celebrations", URL: https://weblog.rubyonrails.org/2007/1/19/rails-1-2-rest-admiration-http-lovefest-and-utf-8-celebrations/
9. Hiroyasu Shimoyama "ApplicationModel のある風景", URL: https://speakerdeck.com/hshimoyama/rails-with-applicationmodel

参考文献

10. Seiei Miyagi "ActiveRecordのモデルが1つだとつらい", URL: https://qiita.com/hanachin_/items/ba1dd93905567d88145c

11. Bryan Helmkamp "7 Patterns to Refactor Fat ActiveRecord Models", URL: https://codeclimate.com/blog/7-ways-to-decompose-fat-activerecord-models/

12. 諸橋恭介 "フォームオブジェクトとの向き合い方", URL: https://speakerdeck.com/moro/grow-form-objects-up

13. Shinichi Maeshima "レールの伸ばし方", URL: https://speakerdeck.com/willnet/rerufalseshen-basifang

ご清聴ありがとうございました

Appendix

6ページ掲載の各資料について

- [上段左] 参考文献13と同じ
- [上段中央] "「Railsでまだ消耗しているの?」—僕らがRailsで戦い続ける理由—", URL: https://speakerdeck.com/toshimaru/why-we-use-ruby-on-rails
- [上段右] 森久太郎 "Microservices Maturity Model on Rails", URL: https://speakerdeck.com/qsona/microservices-maturity-model-on-rails
- [下段左] Tomohiro Hashidate "Realworld Domain Model on Rails", URL: https://speakerdeck.com/joker1007/realworld-domain-model-on-rails
- [下段中央] 参考文献12と同じ
- [下段右] 参考文献9と同じ