Simple組み合わせ村から
大都会Railsにやってきた俺は

Sat, 18 Jan 2025 - 東京Ruby会議12

@moznion

Taiki Kawakami @moznion

SmartBank, Inc.

Software Engineer

@moznion @moznion

@moznion

Railsとの出会い


Railsによる
アジャイル
Webアプリケーション開発
Dave Thomas and David Heinemeier Hansson
with Leon Breedt, Mike Clark, Thomas Fuchs, and Andreas Schwarz 著
前田修吾 監訳
The Pragmatic Programmers
Agile Web Development with Rails
RAIL ROAD CROSSING
Ohmsha

- Rails 1〜3をちょっと触っていた
- あくまでHobby Use

- SmartBankへ転職
- 2024年から本格的に商用のコードを触りはじめる
- 現在はRailsで開発業務をやっている
  - Railsのアップグレードもやっている
  - そろそろ8系に上がる予定

（かつての）Railsに対する漠然とした抵抗感

- フルスタック故の鈍重な印象
- 規約が強すぎる
- アップグレードが大変そう……
- などなど

- フルスタック故の鈍重な印象
- 規約が強すぎる
- アップグレードが大変そう……
- などなど

当時はソフトウェアエンジニアとしての経験が浅く、
不当なバイアスであったことがこの発表準備で再認識された

Rails界に来るまでの経験
～シンプル組み合わせ～

- Perl: Sledge, Amon2
- Node: Express
- Go: net/http
- Java: Spring Boot, Play 2（これらはフルスタックでは？？）

そんなこんなでフルスタックや「設定より規約（CoC）」哲学のフレームワークが長らく食わず嫌いっぽい感じに……

本トークの目的

様々なフレームワークを触ってきて経験が積めた

=> 一定フェアに評価ができるはず

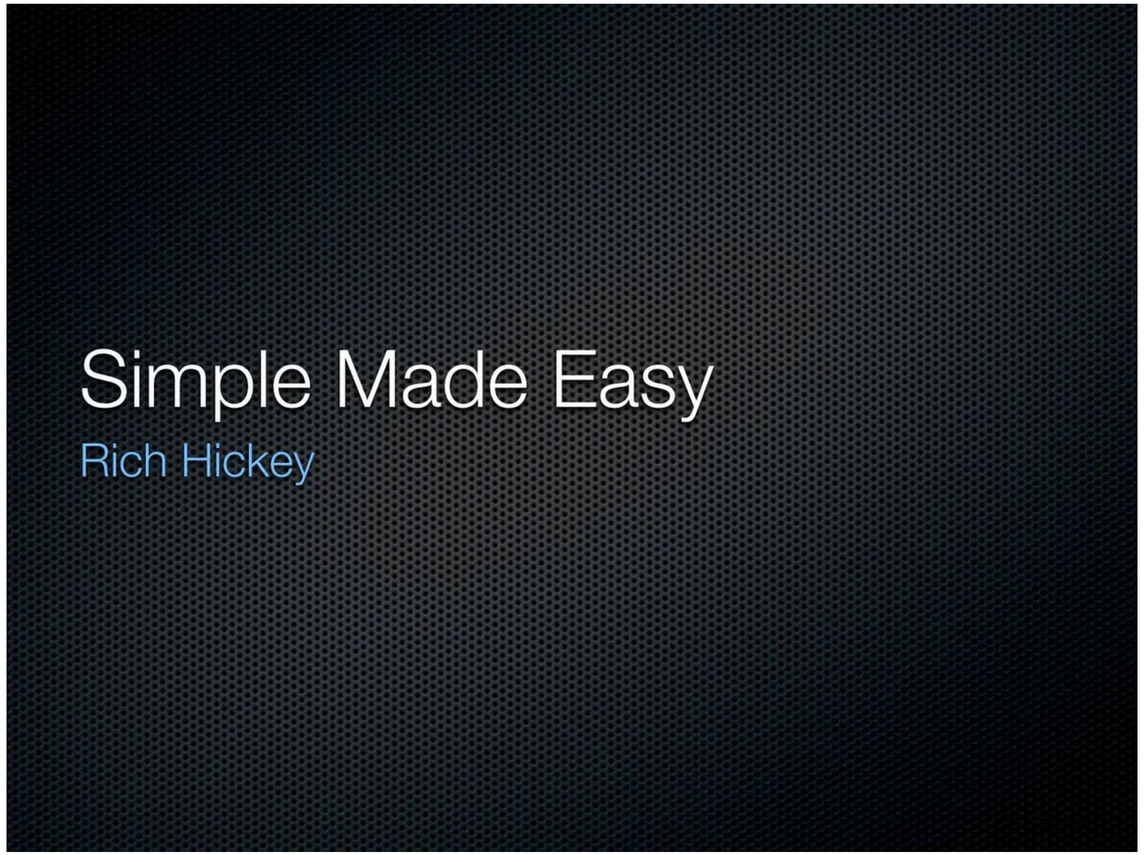
- Rails（や、他のフルスタック）がソフトウェアに与える作用
- シンプルなコンポーネントの組み合わせがソフトウェアに与える作用
- かつての技術選定・今の技術選定が正しかったのかどうか

- Rails（や、他のフルスタック）がソフトウェアに与える作用
- シンプルなコンポーネントの組み合わせがソフトウェアに与える作用
- かつての技術選定・今の技術選定が正しかったのかどうか

このあたりを評価したい

"Simple" と "Easy"

- Clojureの作者であるRich Hickey氏が提唱した "Simple Made Easy" が出自
  - https://www.infoq.com/presentations/Simple-Made-Easy/



Simple

・単一の責務やロールを持つ
・小さい構成要素で成立
 => 結合度・依存度が低い
・仕組みが明解
 => マジカル度が低い
 設計意図を直接把握しやすい

Easy

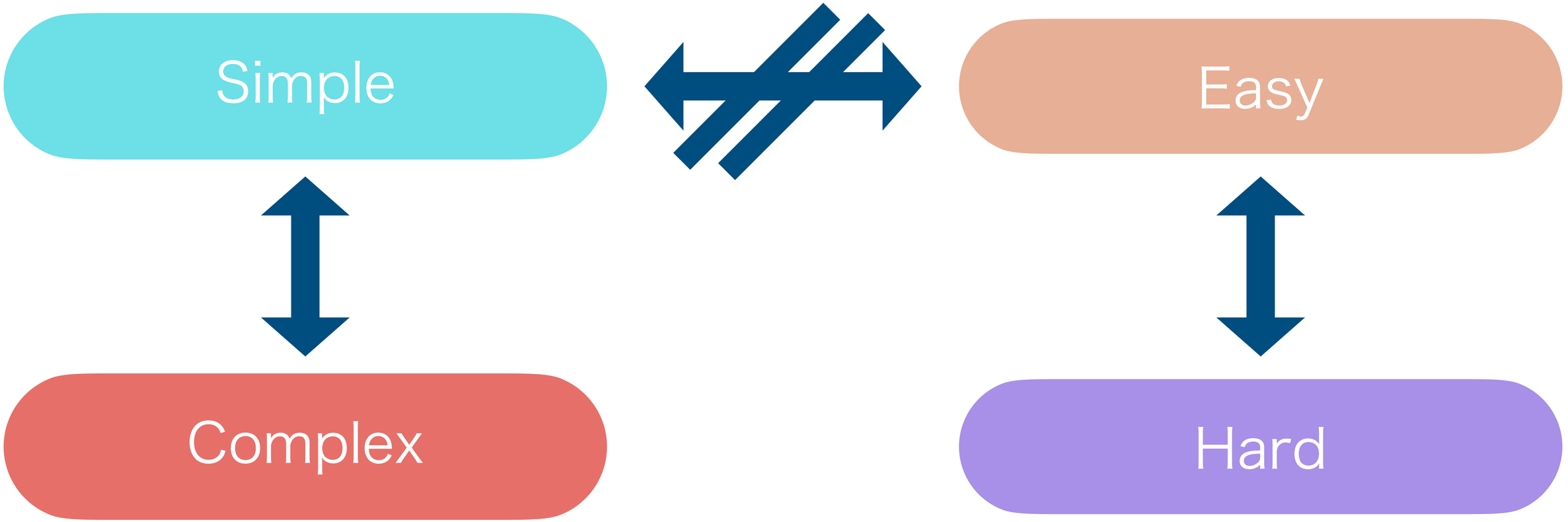
・(何かするにあたって)易しい
・機能を "良い感じ"に用意してくれる
・設定や追加実装が少ない
 => 最初の立ち上がりがクイック

SimpleとEasyは対立構造ではない

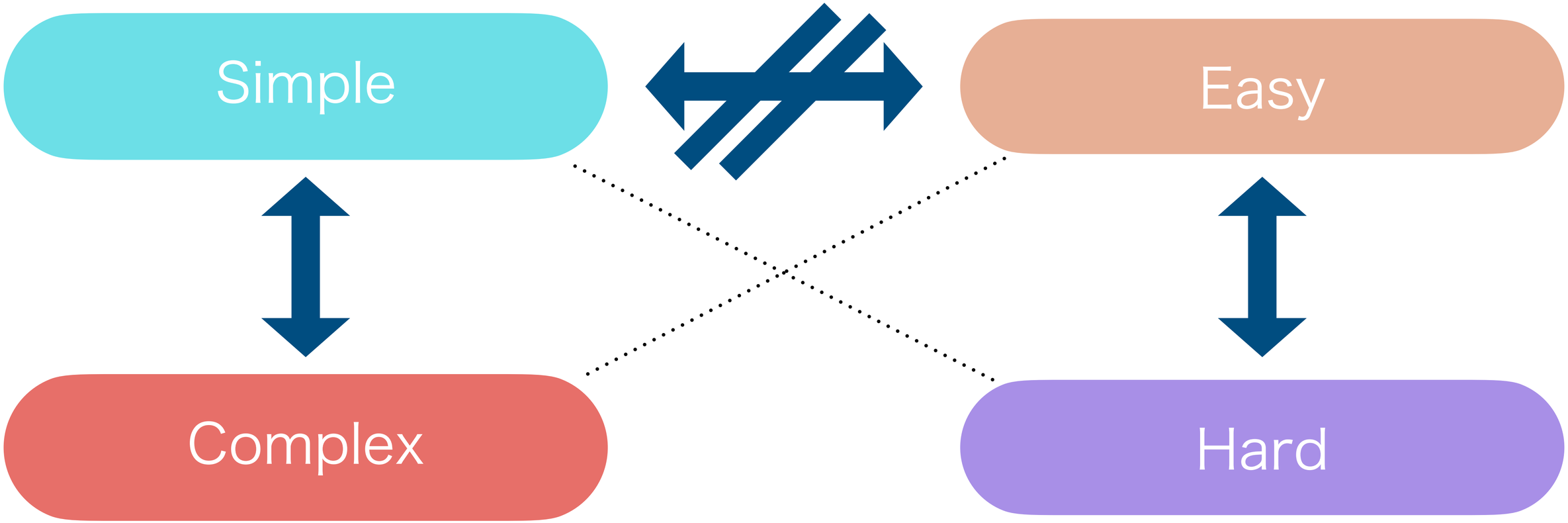
Simple

Easy

SimpleとEasyは対立構造ではない

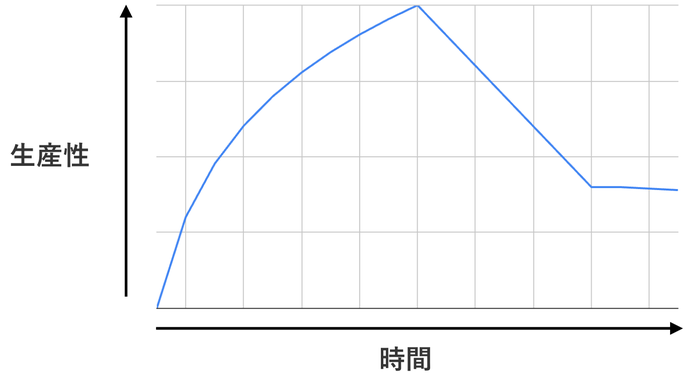


SimpleとEasyは対立構造ではない



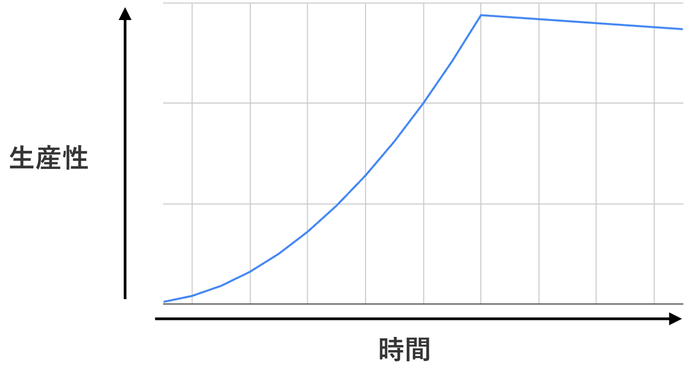
Easy but Complex

・短期的には生産性が高い
=> 複雑であることにはある程度目をつむれる
・中長期的には生産性が低下する（可能性がある）

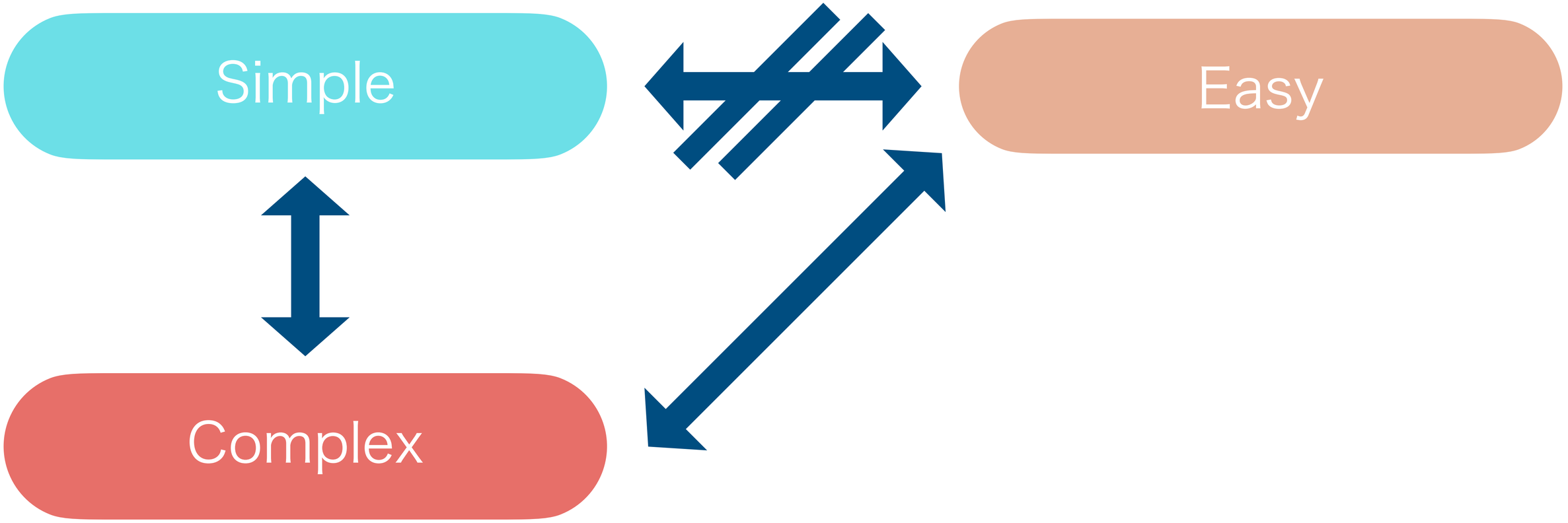


Simple but Hard

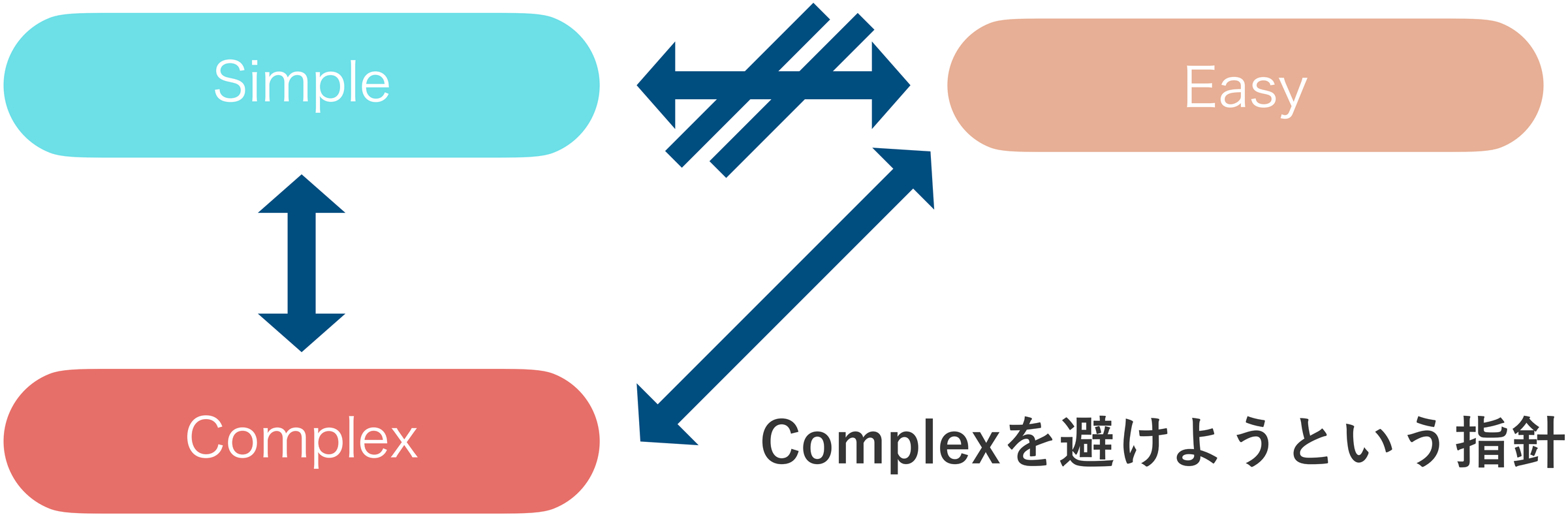
・短期的には生産性が低い
=> Simple/Primitiveなものを組み合わせて問題に適用するため始動までが重い
・中長期的には生産性が高い状況を維持できる（はず）



SimpleとEasyは対立構造ではない



SimpleとEasyは対立構造ではない



Complexを避けようという指針

SimpleとEasyは対立構造ではない

simple組み合わせ村から大都会Railsにやってきた俺は

Ruby on Railsといえばご存知フルスタックフレームワークであり、昨今ではThe One Person Frameworkでおなじみとなっています。
発表者が本格的にRailsアプリを触りはじめたのは去年からです。それまではPerlやGoなどの上でミニマルなフレームワークを用いてwebアプリケーションを開発、あるいは各プロジェクトごとにシンプルなコンポーネントを組み合わせることで専用のフレームワークを作ってその上でアプリケーションを開発していました。
一方Railsはなんでも持っているフルスタックフレームワークであり、いわゆる “easy” 寄りのフレームワークとされている認識です。
つまりこれは巷でよく対比として用いられる “Simple vs Easy” の構図であると捉えられると思います。もちろんsimpleにもeasyにも双方にpros/consがあり一概にどちらが良いと断じることができるものではありませんが、easy寄りのRailsに触れ親しむことでなんとなくその利点について実感を持てるようになってきました。
一方、Webアプリ開発はそういったSimpleとEasyのみで二分し判断できるものではないと考えています。本発表では様々な対比観点からフルスタック/ミニマルの双方を検討し、Webアプリケーション開発に及ぼす影響について考察したいと考えています。

---

moznion

@moznion　@moznion

SimpleとEasyは対立構造ではない

simple組み合わせ村から大都会Railsにやってきた俺は

Ruby on Railsといえばご存知フルスタックフレームワークであり、昨今ではThe One Person Frameworkでおなじみとなっています。
発表者が本格的にRailsアプリを触りはじめたのは去年からです。それまではPerlやGoなどの上でミニマルなフレームワークを用いてwebアプリケーションを開発、あるいは各プロジェクトごとにシンプルなコンポーネントを組み合わせることで専用のフレームワークを作ってその上でアプリケーションを開発していました。
一方Railsはなんでも持っているフルスタックフレームワークであり、いわゆる“easy”寄りのフレームワークとされている認識です。
つまりこれは巷でよく対比として用いられる“Simple vs Easy”の構図であると捉えられると思います。もちろんsimpleにもeasyにも双方にpros/consがあり一概にどちらが良いと断じることができるものではありませんが、easy寄りのRailsに触れ親しむことでなんとなくその利点について実感を持てるようになってきました。
一方、Webアプリ開発はそういったSimpleとEasyのみで二分し判断できるものではないと考えています。本発表では様々な対比観点からフルスタック/ミニマルの双方を検討し、Webアプリケーション開発に及ぼす影響について考察したいと考えています。

!?

---

moznion

@moznion　@moznion

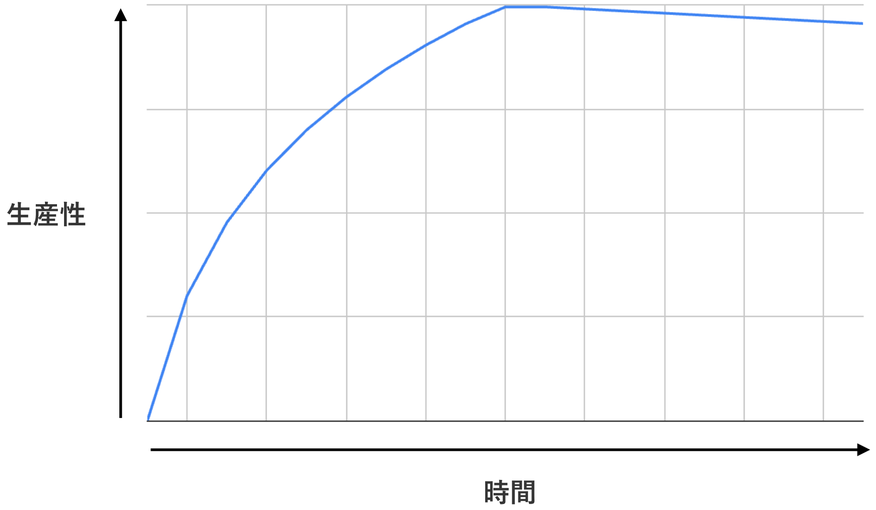

れる “Simple vs Easy” の構図であると
があり──概にどちらが良いと断じるこ

コラ！！！！！！！！！！！

- すみません、不勉強でした……
- [強調] SimpleとEasyは対立構造ではない！！
  - 複雑さを避けましょう、という指針
- Simple and Easyが理想

こうなってほしい

- すみません、不勉強でした……
- [強調] SimpleとEasyは対立構造ではない！！
  - 複雑さを避けましょう、という指針
- Simple and Easyが理想

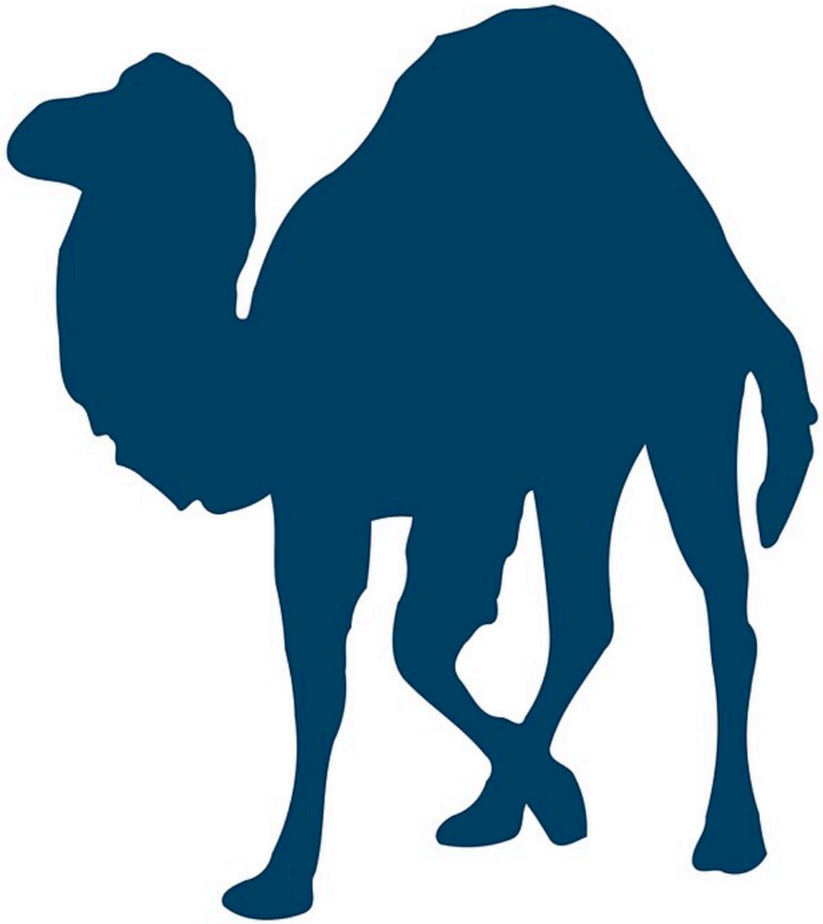


フルスタックフレームワークと
シンプル組み合わせを比較するとどうか？

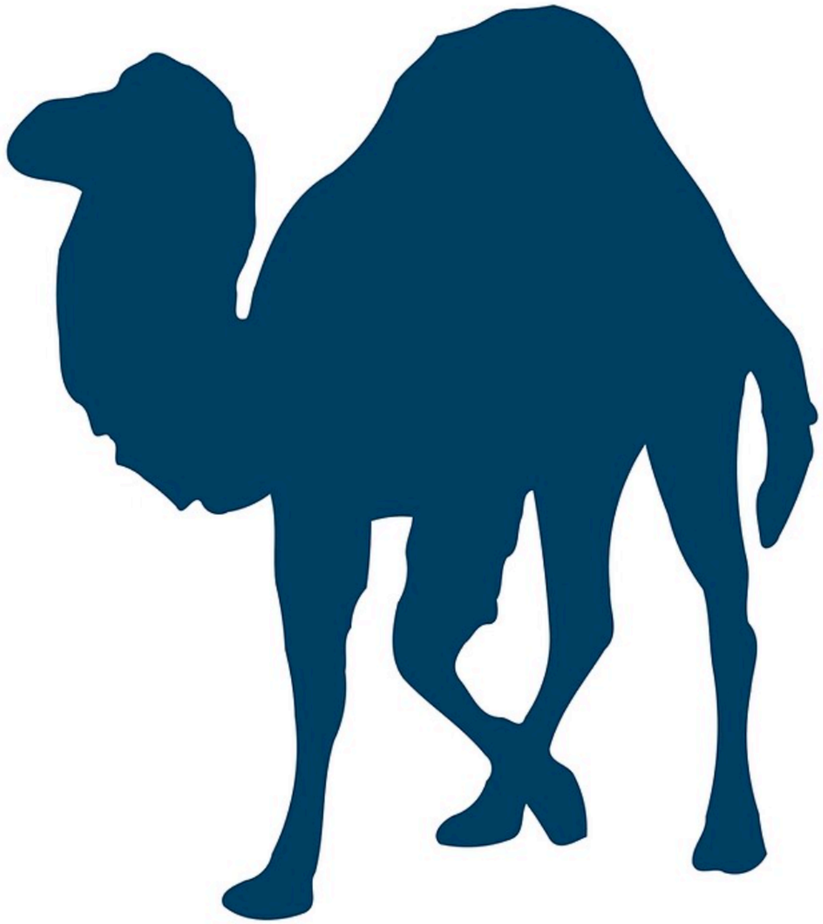
そもそもシンプル組み合わせとは何か？

～10数年前の日本における一部のWebアプリの作り方を添えて～

そもそもシンプル組み合わせとは何か？



Perl!!!



- パーツを組み合わせてやっていくスタイルが強かった
  - 薄いフレームワークが人気だった
    - goの世界観に若干近い（諸説ある）(†1)
- （念の為）Perlにもフルスタックフレームワークはある
  - Catalyst
  - Mojolicious

†1: とはいえGoはTLSが言語コアに入っていたりしてリッチですからね……

パーツ例

- PSGI layer
- Router
- Controller
- ORM
- Class Library (†1)
- Template Engine (JSON)
- Logger
- Security
- Middleware (Session Store, and etc)
- etc...

†1: https://speakerdeck.com/moznion/perl-5-oopji-gou-30nian-shi-perl-5-s-oop-mechanism-past-30-years

パーツ例

- PSGI layer
- Router
- Controller
- ORM
- Class Library (†1)
- Template Engine (JSON)
- Logger
- Security
- Middleware (Session Store, and etc)
- etc...

色々組み合わせる楽しさ（諸説あり）

†1: https://speakerdeck.com/moznion/perl-5-oopji-gou-30nian-shi-perl-5-s-oop-mechanism-past-30-years

- Perlの言語コアに対する考え方
  - 言語自体の実装にアプリケーション的な機能をあまり追加しない
  - その代わりのcore moduleという概念 (†1)

†1: 例えばJSONも言語実装には入っておらず、3rd partyのJSON::PPライブラリがcore moduleとしてバンドルされている

- 当時の状況
  - クラウドコンピューティングの夜明けみたいな時代
  - オンプレが依然元気。その上でハイトラフィックを捌くプレイヤーが多かった
    - 細やかなパフォーマンスチューニングが必要
    - 速いパーツをどんどん使っていく必要があった
      - マイクロチューニングではある
      - それすら効いてくる世界観

一方フルスタックフレームワークとは

- 必要なものが全部入り
- 先に挙げたパーツの機能がほとんど具備されている
  - router, controller, ORM, template engineなどはもちろん
  - RDBMSのmigration機能などもあったりする

Action Cable
ActionController
ActionDispatch
ActionMailbox
ActionMailer
ActionText
ActionView

ActiveJob
ActiveModel
ActiveRecord
ActiveStorage
ActiveSupport
etc...

- battery-included
  - RDBMSによってCRUDするようなアプリは即座に作りはじめられる
  - 宣言的に書くだけで動く
  - 普通に考えるとめちゃくちゃ難しいDBの継続的migrationなんかもある
    - ActiveRecordは本当に偉大
- スケルトンやボイラープレートが良い感じに生成される
- 統一的なツールチェインが整備されている（e.g. bin/rails）

Easy, Simple, Complex, and Hard

💎 EasyなRailsはComplexなのか？ => 所によりYes

🐫 Simple組み合わせはHardなのか？ => 所によりYes

💎 EasyなRailsはComplexなのか？ => 所によりYes

🐫 Simple組み合わせはHardなのか？ => 所によりYes

しかしSimpleとEasyは両立できてほしい！

フルスタックとSimple組み合わせを
別の切り口で対比してみる

	Rails	Simple組み合わせ
一貫性 vs 柔軟性	一貫性寄り	柔軟性寄り
暗黙的 vs 明示的	暗黙的寄り	明示的寄り
包括的 vs ミニマル	包括的	ミニマル
強思想 vs ノンポリ	強思想寄り	気持ちが必要
ラピッド開発 vs ロングターム運用	どちらも	どちらも

Rails
(一貫性寄り)

- Railsの規約によって統一的な構成・構造が半強制される
- Rails Wayから外れると大変
- 柔軟性が無いわけではない
  => ライブラリは選べる && 自分で作れる

Simple組み合わせ
(柔軟性寄り)

- 用途に応じてコンポーネントを差し替えれば良い
  => 最悪作ればよい
- 基本的にやりたいことはなんでもできる
- 一貫性を持たせるには強い気持ちが必要

Rails
(暗黙的寄り)

- Convention over Configuration
- 暗黙的な挙動が多い
  => マジカル
- 初期の学習曲線が急
- マスターすると高速に開発できる
  => いちいち書く必要がない

Simple組み合わせ
(明示的寄り)

- 基本的にすべて書かれているため明示的
- 暗黙的機構であってもプロジェクト内部に書かれているはず
- 明示的に書くぶん手数が増える
- 学習曲線は急峻ではないもののプロジェクトごとに学ぶ必要がある

Rails
(包括的寄り)

- 全てが入っていることにより
  選択をする労力から解放される
- presetの構成要素間でのシナジー
- 小さなものを作る時にオーバー
  キルになる可能性

Simple組み合わせ
(ミニマル寄り)

- 所望の技術選定ができる
- 結合も自分で組める
  (柔軟性・明示的にも係る)
- パフォーマンスチューニングも
  やりやすい

Rails
(強思想寄り)

- MVC + ActiveRecordといった
  強い初期設計
- DHHの強い思想
  => RoR Guidesという経典
  利用者の多さによる思想の強化
- 思想外のユースケースの対応

Simple組み合わせ
(気持ちが必要)

- シンプル組み合わせは可能性が無限
  => ルールも無限（無いとも取れる）
  ここに思想を持てるかどうか?
- ユースケースが初期の強い思想に
  束縛されない

Rails
(どちらも)

・小さなアプリから大きなアプリまで
・The One Person Framework
（from hello world to IPO）
=> IPOはすぐできるもんではない
長い間メンテするという意識?
・バージョンアップ大変問題
=> 最近は大変さが軽減されてそう

Simple組み合わせ
(どちらも)

・小さなものはシンプルにクイックに作れる
・大きなものをrapidに作るには初動が重い
=> "way" を作る必要がある
・ライブラリのライフサイクルの枷
・ドメインに沿った最適化ができる
・コンポーネントのリロケーションがやりやすい

	Rails	Simple組み合わせ
一貫性 vs 柔軟性	一貫性寄り	柔軟性寄り
暗黙的 vs 明示的	暗黙的寄り	明示的寄り
包括的 vs ミニマル	包括的	ミニマル
強思想 vs ノンポリ	強思想寄り	気持ちが必要
ラピッド開発 vs ロングターム運用	どちらも	どちらも

	Rails	Simple組み合わせ
一貫性 vs 柔軟性	一貫性寄り	柔軟性寄り
暗黙的 vs 明示的	暗黙的寄り	明示的寄り
包括的 vs ミニマル	包括的	ミニマル
強思想 vs ノンポリ	強思想寄り	気持ちが必要
ラピッド開発 vs ロングターム運用	どちらも	どちらも
	Easy (but complex?)	**Simple (but Hard?)**

	Rails	Simple組み合わせ
一貫性 vs 柔軟性	一貫性寄り	柔軟性寄り
暗黙的 vs 明示的	暗黙的寄り	明示的寄り
包括的 vs ミニマル	包括的	ミニマル
強思想 vs ノンポリ	強思想寄り	気持ちが必要
ラピッド開発 vs ロングターム運用	どちらも	どちらも
	Easy (but complex?)	Simple (but Hard?)

やはり両立しないのか…?

フレームワークが与えるソフトウェアへの作用

- RailsはMVCを強くアフォードしている
  - それ以外のアーキテクチャを採用するにあたってのハードルが生じる
  - i.e. レールを外れる、脱線……
- その他にもIDLのフルサポートを受けた構成など
- 自分で作るぶんにはこのあたりの自由度は高い

- Railsは基本的にHTTP (L4: TCP) 上で動くアプリケーションを主眼としている(†1)
  - 別のプロトコルを利用したい場合の選択肢はかなり制限される
- 自分で作るならこのへんはやりたい放題
- 近い話題として: アプリを動かすためのデプロイ先の選択にも影響する

†1: WebSocketなどの例外はある

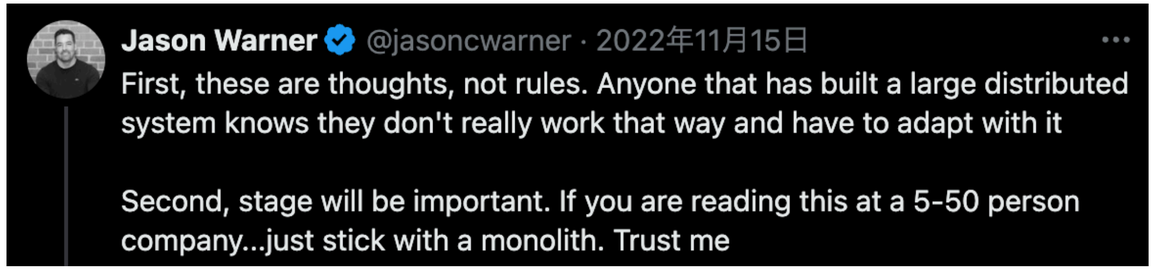
- フレームワークを乗せ換えることは滅多に無いが、実際は色々ある
  - 例:
    - Play 2を運用している最中のAkkaのライセンス変更 (†1)
    - Expressの開発が一時期停滞していた事例 (†2)
- 組み合わせで作っていると、いきなり危険な状態になることはない
  - 一方で個別のライブラリで同様の事例が発生する可能性はある
    - 突然EOLを迎えるなど

†1: Play 2はライセンス的に特例で免責されていた + Pekkoが出てきたというのもあり難は逃れた
†2: 今はぜんぜん元気

- 大きなフレームワークのMajorアップグレード
  - やはり大変。ちゃんと継続してやらないと廃墟になる。
  - とはいえ最近のアップグレード難易度はそこまで高くない印象
- 小さなコンポーネント・ライブラリを継続的にアップグレード
  - 徐々にアップグレードできるので影響の局所化が容易
  - 一方で依存管理が複雑
    - 或るライブラリAのアップグレードがライブラリBに影響したり……
    - 依存のアップデートをウォッチする必要がある

フレームワークが与える開発組織への作用

- Railsはどちらかというとmonolithic寄りの印象
- microservicesの各コンポーネントはミニマルなフレームワークで作られがち？
  - とはいえRailsでmicroservicesをやっても良いと思う
- ここについては特にフレームワークによる大きな影響はなさそう
- 余談: Mega Rails UserことGitHubのCTO Jason Warner氏の示唆的発言



参照: https://x.com/jasoncwarner/status/1592227285024636928

- Railsであればコモディティな情報がWebに大量にあるので学習リソースが潤沢
  - 一方でどう検索すれば良いんだ、みたいな「規約」はある
    - 今ならLLMに聞くという方法がある
  - 技術キャッチアップに関してもRailsの情報をメインで追うと良い
  - とりあえずRailsを知っていれば他のRailsプロジェクトにも応用ができる
    - 最初の急な学習曲線だけを乗り越えれば……

- シンプル組み合わせはその組み合わせに依存するので共通の情報が少ない
  - ある程度はパターン化しているとは思うのでそこにナレッジはありそう
  - 各パーツ自体はシンプルなはずなので個別に調べることはできる
- 組み合わせ自体は共通化した知識ではない
  - 別のプロジェクトに移ったときにそのまま応用できるものではない
  - LLMに聞くのもコツが必要
  - その一方で「組み合わせ」はフレームワークを作っているようなもの
    - その基礎的な知識は普遍的（なはず）

- Railsは規約により束縛されており、利用者が多く、ベストプラクティスが確立されている
  - 属人性が低い
  - 一方で暗黙的な挙動が多いので、そこに対する理解のしやすさやトラブルシューティングの容易さがトレードオフ
- Simple組み合わせは初期の作成者による属人性が強くなりがち
  - Simpleであるとはいえ作者の暗黙知が入る
  - これをなんとかするのが腕の見せどころではある

- Majorなフレームワークや技術セットは採用しやすい！！！！！
  - これはガチ
  - これはガチ

で、結局SimpleとEasyは両立しないのか？


The One-Person Framework

- とはいえ一人で全部やるなら何使ってもよくない？
- どちらかというとOne PersonというよりもSmall Teamなのではないか
  - 小さな組織やスタートアップではRailsの優位性が高い
    - 必要なものが最初から揃っている => ビジネスロジックに集中できる
    - 強固なRails Wayという思想・指針
    - Active Record
- 「One Personでできることは健康的にスケールする」という説

- RailsがEasyであるというのは何なのか
  - 一定の制限（規約）があることで作りたいものを容易（クイック）に実現できる
  - 「作りたいもの」の本質と関係ない部分は隠蔽されている
    - 隠蔽されている部分が複雑であることは構わない
- ビジネスロジックから複雑性が排除されていれば良い（そこに集中する）
  - ＝＞ 作るための仕組みはEasy、ビジネスロジックはSimple

とはいえ隠蔽されている部分（i.e. 実装）を読みに行くことが上級者になるほど増える気がする。
上級者にとっては逆にeasyではないのではないか？

Simple組み合わせ村

- 必要なものだけを組み込める良さ
- ベストプラクティスを作れる == 自分達に最適化できる
  - レールを外れることを恐れなくて良くなる
  - 最適化した先にEasyがある
- コンポーネント指向を養うことができる
  - 使用・作成するライブラリに対する審美眼であったり、
    自分が或る機構を作成するために何をどうすべきかという思考が養われる
  - フルスタックフレームワークを使っていてもライブラリ選定はする
    - そもそもフルスタックフレームワークの選定もするので……

- システム全体のアーキテクチャを考え抜いた上でのベストを探求することができる
  - 例
    - システムのアーキテクチャ（e.g. microservices）
    - デプロイ先
  - 最大公約数的なEasyではなく、ドメインに特化したEasyをSimpleとともに得ることができる
    - そこに至るまでの道はHardかもしれない……
    - HardはComplexよりマシなので頑張ろう！！

以下、個人的なフィーリング

- 仕事でやるならフルスタックでもミニマルでもどっちでも良い
- フルスタックのほうがどちらかというとやりやすいかも
  - クオリティコントロール
  - チームビルディング
  - とはいえ、すぐ手の届く範囲に望む挙動を実現するコードが明示的に書かれていて欲しい感覚は強くある……
- この感覚に得るには実際に商用プロダクトをやってみないとわからない点だった

- 一方、シンプルを組み合わせるのは楽しい！！！
- 自分好みの技術選定ができる
- 自分の考えるベストなアーキテクチャを実現できる
  - その上で所望のアプリケーションを動かす快感
- モノ作りの本質的体験（おたのしみ要素は強い）

- それはそうとして多様性があることは良いことだと思う
  - 今のRubyはひいき目に見てもRailsの一強体制なように思う
  - Sinatra等はあれど、シンプルを組み合わせる風潮がやや弱い？
- 他の言語だと新機軸のORM (†1) が出てきたりしている
  - そのあたりをフルスタックフレームワークが組込もうとすると大変そう
- JavaScript界にはhonoがやってきたりして元気がありよさそう

†1: PrismaやDrizzleなど

まとめ

- フルスタックフレームワークを利用するのもSimple組み合わせを利用するのも「複雑性を排除しつつ所望のドメインを満足する」という目的のもとであればどちらも正解！！！！！
  - それぞれがソフトウェアに及ぼす効果はあるが、目指すゴールは同じ。
  - 特定の手法に固執せず、適材適所に使い分けられるのが良いのでは(†1)
- 「ドメイン実現のための複雑性を可能な限り排除する」とにかくこれに尽きる

†1: チーム組成や採用の作戦にも依存する話題ではありますが