2021/12/16 RubyWorld Conference 2021 | Shu OGAWARA (@expajp)

Railsバージョンを倍にしたサービスのそれまでとそれから

2020年7月、所属しているリンカーズの
Twitterアカウントでこんなツイートを行いました

Linkers Tech
@Linkers4System

たった今、Linkers.net のRailsのバージョンが5.2から6.0に上がりました🎉

3ヶ月前の5/18までRails3だったサービスがとうとう最新版の6までアップグレードされ、3年前のリファクタリングから始まったプロジェクトがついにゴールを迎えました！

2020年、リンカーズは一気に

Rails 3 → 6 へとバージョンアップを果たしました🎉

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also compare across forks.

base: v3.2.22.5 ← compare: v6.0.3.2

Commits 5,000+ | Files changed 4,820 | Commit comments 5

Showing **4,820 changed files** with **419,383 additions** and **185,343 deletions**.

なぜ2020年までRails3.2のままだったのか？

どのような経緯でリファクタリングPJを始めたか？

どうやってリファクタリングを進めたのか？

Agenda

もくじ

- 自己紹介
- リンカーズの紹介
- なぜリンカーズはRails3.2のままだったのか？
- リファクタリングPJ開始の経緯
- リファクタリングの進め方
- Railsアップグレード

自己紹介

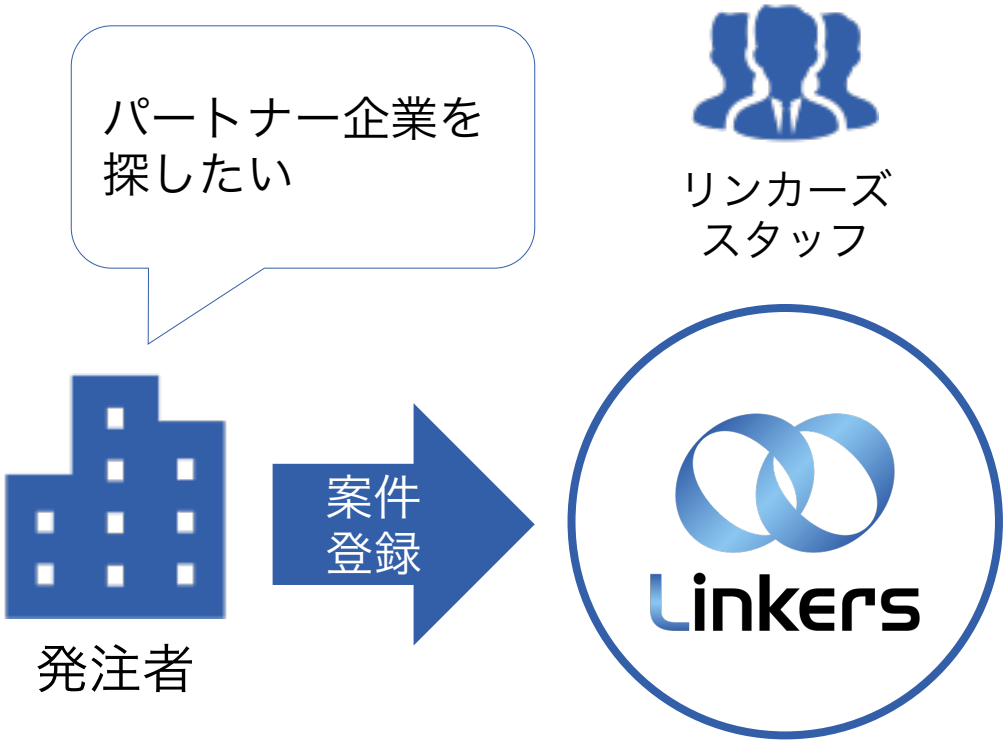
- Shu OGAWARA(@expajp)
  - 「えくすぱ」と読みます
  - TwitterもGitHubもこのアカウントです
- リンカーズ株式会社
- Engineering Manager Meetup Staff.

リンカーズの紹介

Linkers

ものづくり業界で
パートナー企業をマッチングする

ものづくり系ビジネスマッチング



ものづくり系ビジネスマッチング

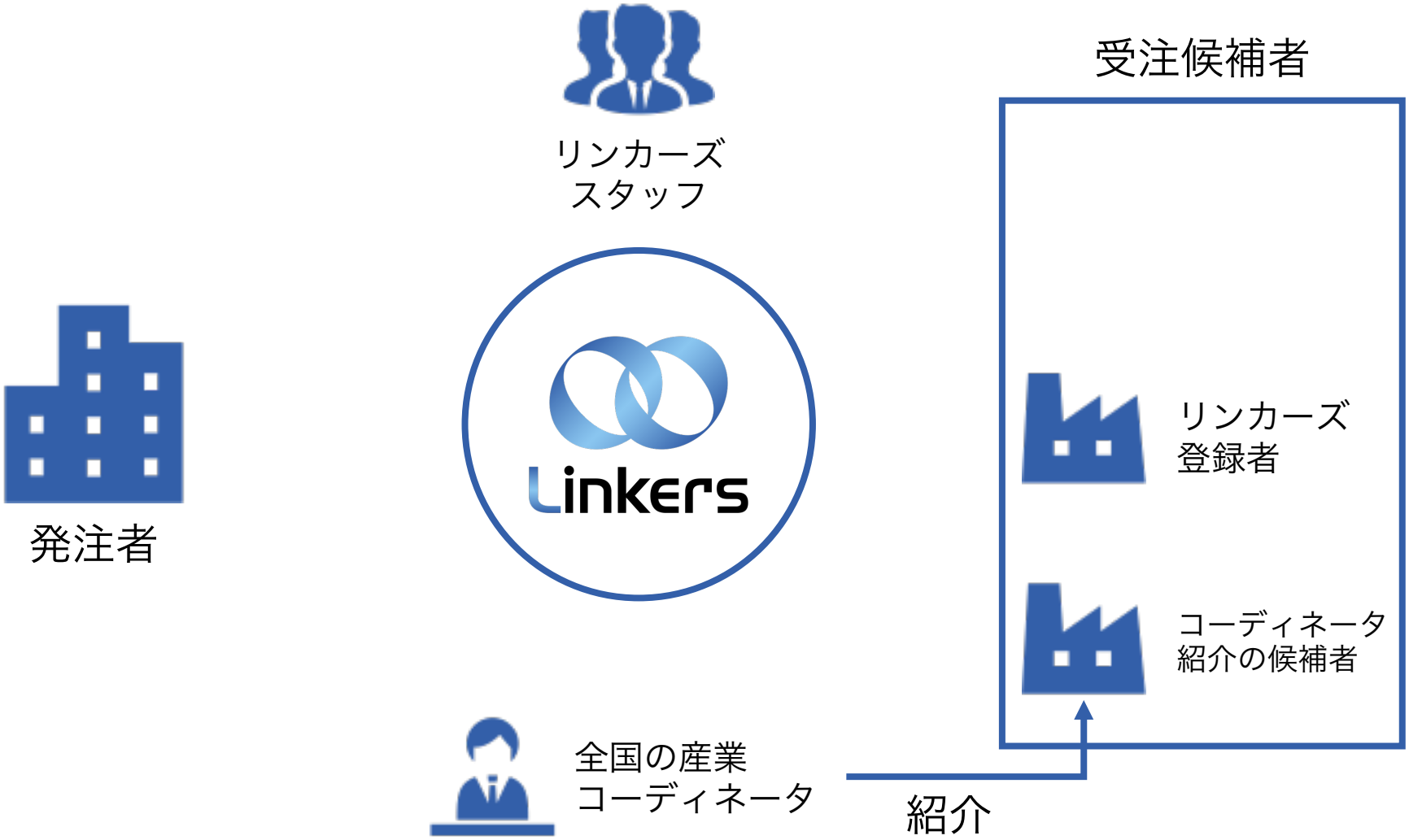
リンカーズ
スタッフ

受注候補者

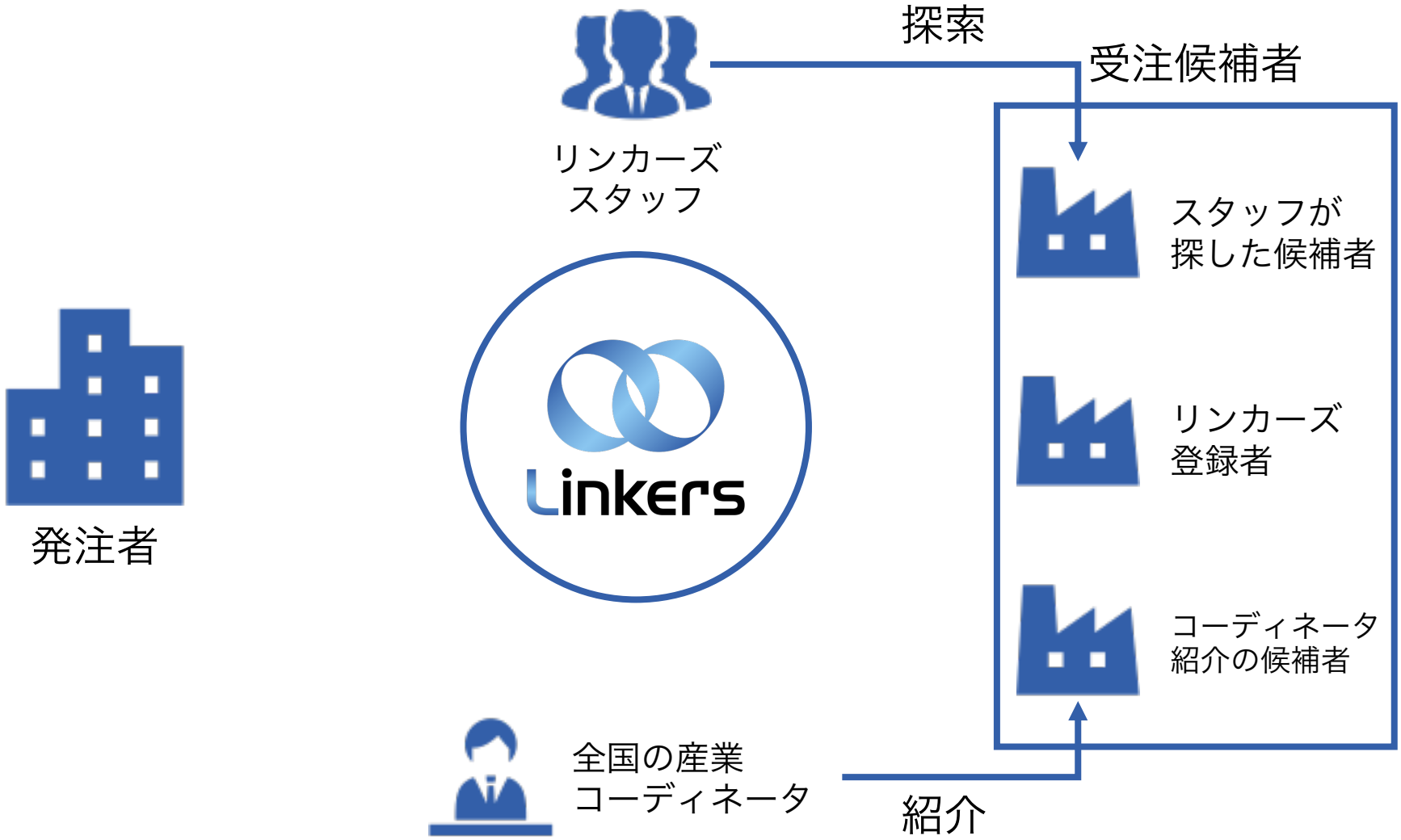
リンカーズ
登録者

発注者

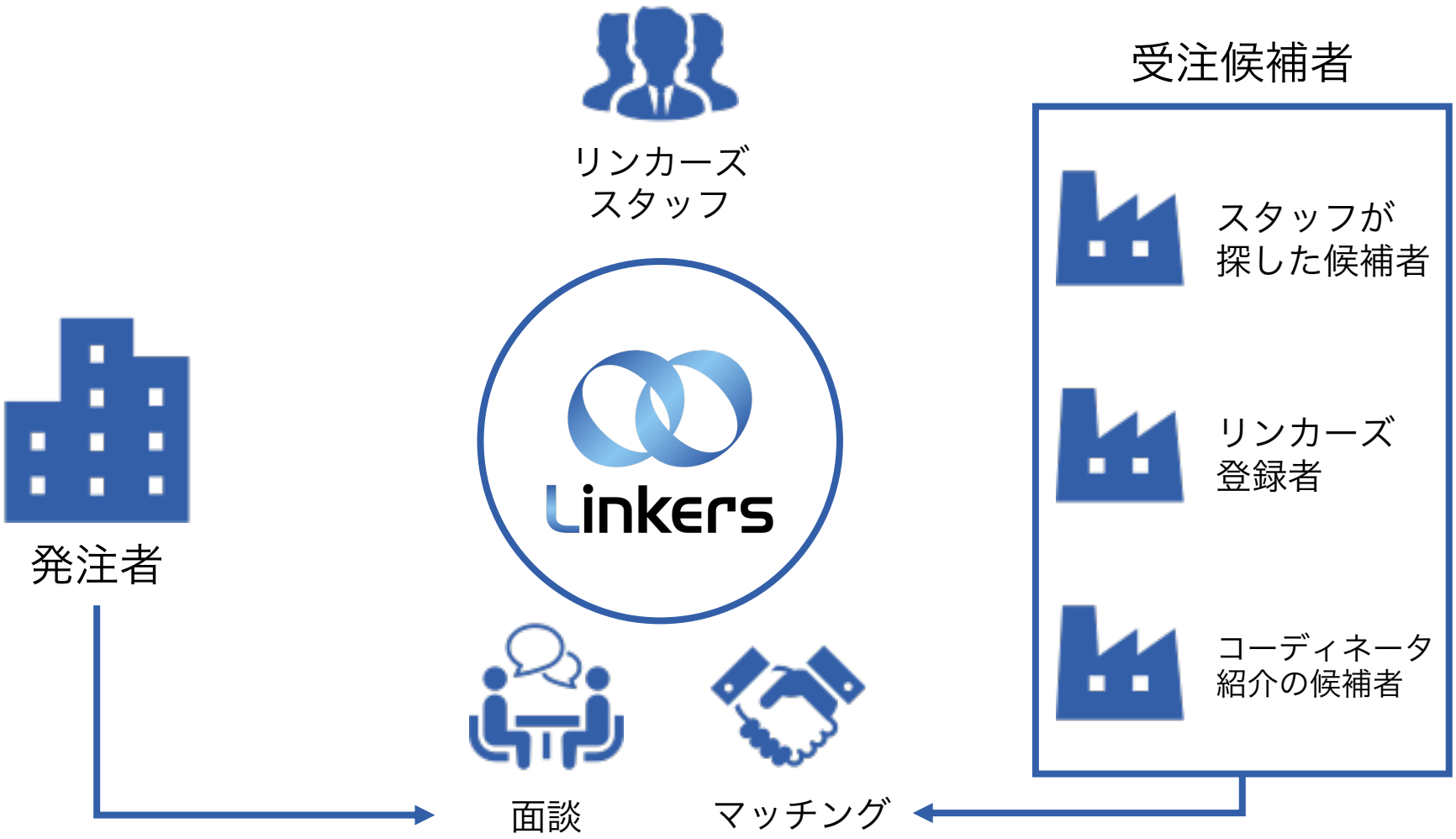
ものづくり系ビジネスマッチング



ものづくり系ビジネスマッチング



ものづくり系ビジネスマッチング



ものづくり系ビジネスマッチング

- システム利用者は主にリンカーズスタッフ
- 現在は「リンカーズ登録者」が増えたため、
  この方々を対象に別サービスも展開中

なぜ2020年までRails3.2のままだったのか？

どのような経緯でリファクタリングPJを始めたか？

どうやってリファクタリングを進めたのか？

なぜ2020年までRails3.2のままだったのか？

どのような経緯でリファクタリングPJを始めたか？

どうやってリファクタリングを進めたのか？

Q. なぜ2020年までRails3.2のままだったのか？

- A. 技術的負債が積み上がっていたから

なぜリンカーズに技術的負債が積み上がったか？

創業と内製化へのシフト

- 2012年、Distty社創業 (現リンカーズ)
- 2年後、内製化へシフト
  - リンカーズの前身「eEXPO」は外注で開発
  - しかし、Product Market Fit(PMF)の模索にはスピード不足

内製化のため、CTOのSさん入社

この時のコードはこんな感じ

- 大量の生SQL
- それがコントローラに書かれている
- ビューにもロジックがある
- そのロジックでロールごとに全く別内容の表示をさせている
- そのロールの認可は独自実装
- テストは書かれていない

※現在は当時のこういったコードは残っていません、念のため

なぜこうなったか

- 社内にエンジニアがいなかったため、技術的な品質チェックができなかった
- 2012年当時は日本語のRailsの知見が少なかった

Railsガイド
日本語版は4.1から

しかし、創業期は開発優先

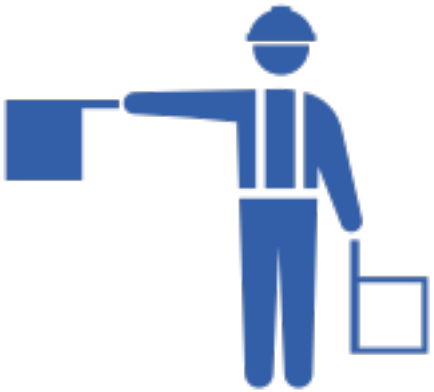
- PMF模索の高速化のため内製化した
- eEXPOですでにビジネスが回っている
- リファクタリングするなら、
  顧客に価値を届けることを後回しにすることになる

最低限の安全性を担保して、開発を継続した

開発を継続した結果

- マッチング機能が好評で順調に成長
  - eEXPOを継続しつつ、
    マッチング特化の「リンカーズ」が
    アプリとして独立
- ビジネスサイドに人が増えてきた

人が増えた結果、開発依頼体制に問題が



- 当時はチャットでSさんに直接依頼
  - 機能の精査が行われない
  - 不要になった機能の棚卸が行われない

→本来なら、人が増えると交通整理をする人が必要

事業の成長は続くものの、
開発の目線では問題点が表面化してくる

現状の確認

- 技術的負債が多い
  ＝**コードの理解が難しい**
- 不要と思われる機能がたくさんある
  ＝**本当に不要かは確認しないとわからない**

両者の組み合わせで起こった問題

- 報告される不具合が本当に不具合か
  ビジネスサイドに確認必須
- 本当に不具合でも、修正が大変
- 修正してる間に次の開発の話が来る

Railsアップグレードができない

- テストがないため、
  アップグレードしても動作保証が
  できない
- 仕様を都度聞いている状態では、
  手動での動作確認もできない

Railsアップグレードはできない状態だが、
リファクタリングに割く時間もない

まとめ：いかにして技術的負債が積み上がってしまったか

1 **創業時に エンジニア不在**

システムの技術的品質をチェックできなかった

2 **創業期は開発優先**

最低限の安全性を担保して、PMF達成を目標に

3 **交通整理が必要になった開発要望**

人が増えると要望の反映にも体制が必要に

なぜ2020年までRails3.2のままだったのか？

どのような経緯でリファクタリングPJを始めたか？

どうやってリファクタリングを進めたのか？

どうやってリファクタリングPJが決まった？

リンカーズをSaaS化して、銀行やグループ
会社に導入したいんだけど、どう思います？

どうやってリファクタリングPJが決まった？

リンカーズをSaaS化して、銀行やグループ
会社に導入したいんだけど、どう思います？

前々からお伝えしていたように、
拡張に向けて、
内部を改善するのが先決です

どうやってリファクタリングPJが決まった？

リンカーズをSaaS化して、銀行やグループ
会社に導入したいんだけど、どう思います？

前々からお伝えしていたように、
拡張に向けて、
内部を改善するのが先決です

前々から何度も聞いてたけど、そんなに大変なんですか
マッチングが好調なので、この路線でスケールできる
よう、時間をかけて内部改善しましょう

経営陣に理解があること、説明することが重要

- 銀の弾丸はない
  - 一般論として、経営判断がないと大規模なリファクタリングは難しい
  - 今回は経営陣もゲームチェンジの必要性を認識していた
  - Sさんが普段から何度も説明して理解を得ており、人柄と腕前への信用を築いていた

なぜ2020年までRails3.2のままだったのか？

どのような経緯でリファクタリングPJを始めたか？

どうやってリファクタリングを進めたのか？

リファクタリングのゴール：メンテナンス可能なシステムにする

- コード面：テストがあるシステムにする
- 機能面：仕様を確定させる
- 両方：使われていないコードを消す

しかし、現在動いているサービスを
止めるわけにはいかない

リファクタリングPJの進め方の案

- 案1: 今のコードにそのままテストを書く
- 案2: 今のコードを捨てて作り直す
- 案3: 機能単位で再設計して作り直す

リファクタリングPJの進め方の案

仕様を確定させない
とテストが書けない

- ~~案1: 今のコードにそのままテストを書く~~
- 案2: 今のコードを捨てて作り直す
- 案3: 機能単位で再設計して作り直す

リファクタリングPJの進め方の案

- ~~案1: 今のコードにそのままテスト~~
- ~~案2: 今のコードを捨てて作り直す~~
- 案3: 機能単位で再設計して作り直す

開発中も既存コードは拡張される

案3: 機能単位で再設計して作り直す

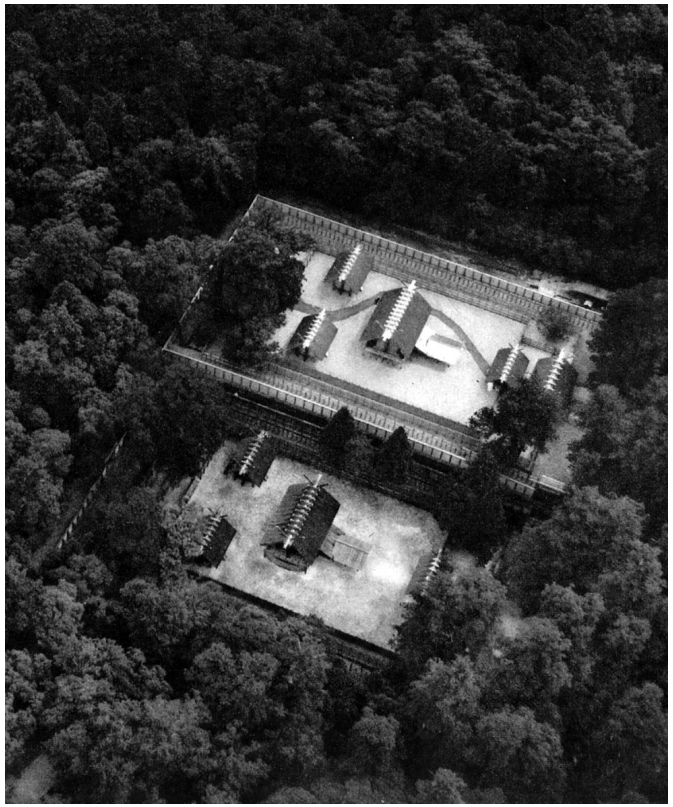
- 新旧コードの共存で考慮することが多い
- 並行するぶん、既存機能拡張も大変

案3: 機能単位で再設計して作り直す

- 仕様確定→再設計の手間が現実的
- 新しいコードにはテストが書ける
- 作り直されなかった箇所が
  使われてないコード

案3: 機能単位で再設計して作り直す に決定

開発の進め方



- MVCぜんぶ遷宮

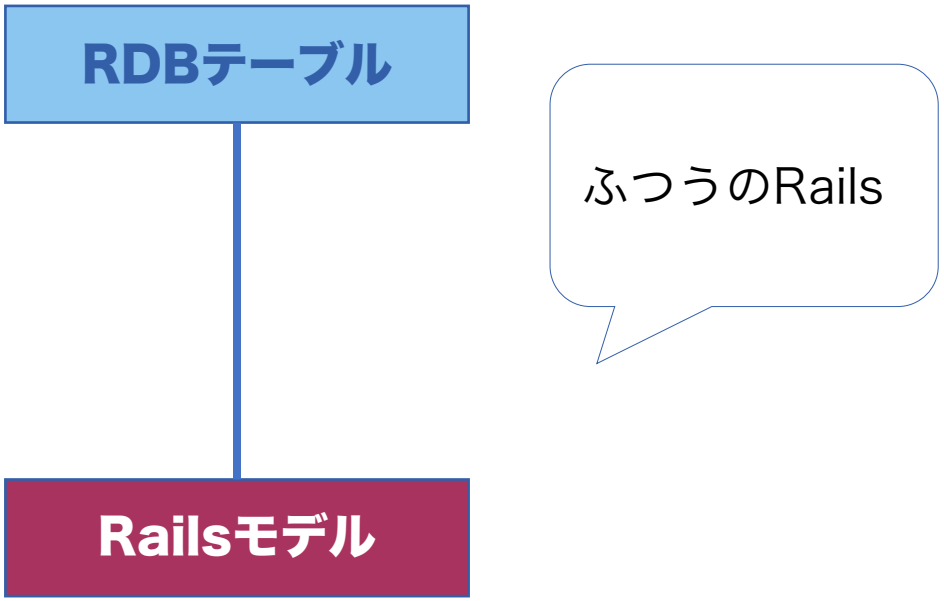
※ソフトウェア開発ではやや誤解されて広まっている気がしますが、
決まった年数＝式年で定期的に作り直すのが式年遷宮なんだそうです

モデル

- 古いモデルと新しいモデルを共存させる

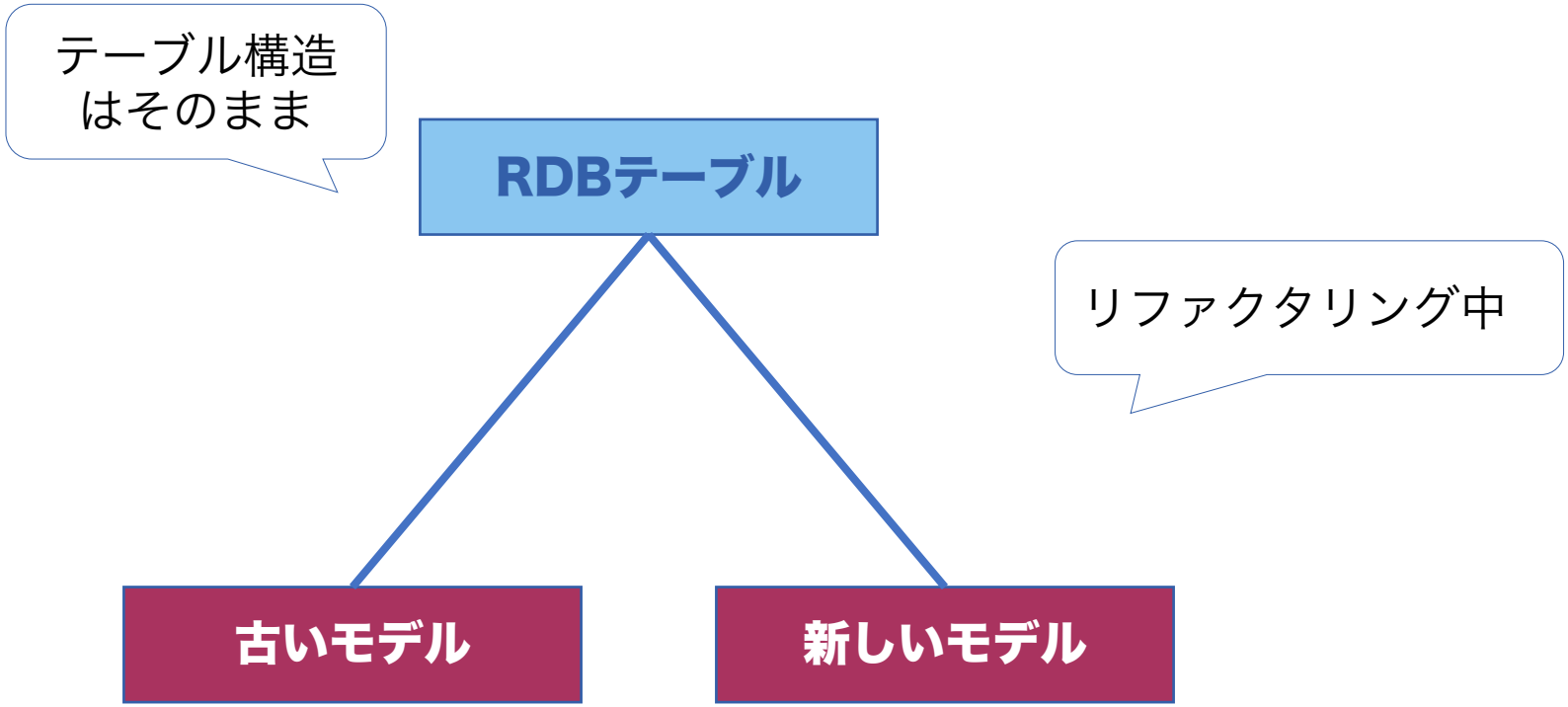
モデル

- 古いモデルと新しいモデルを共存させる



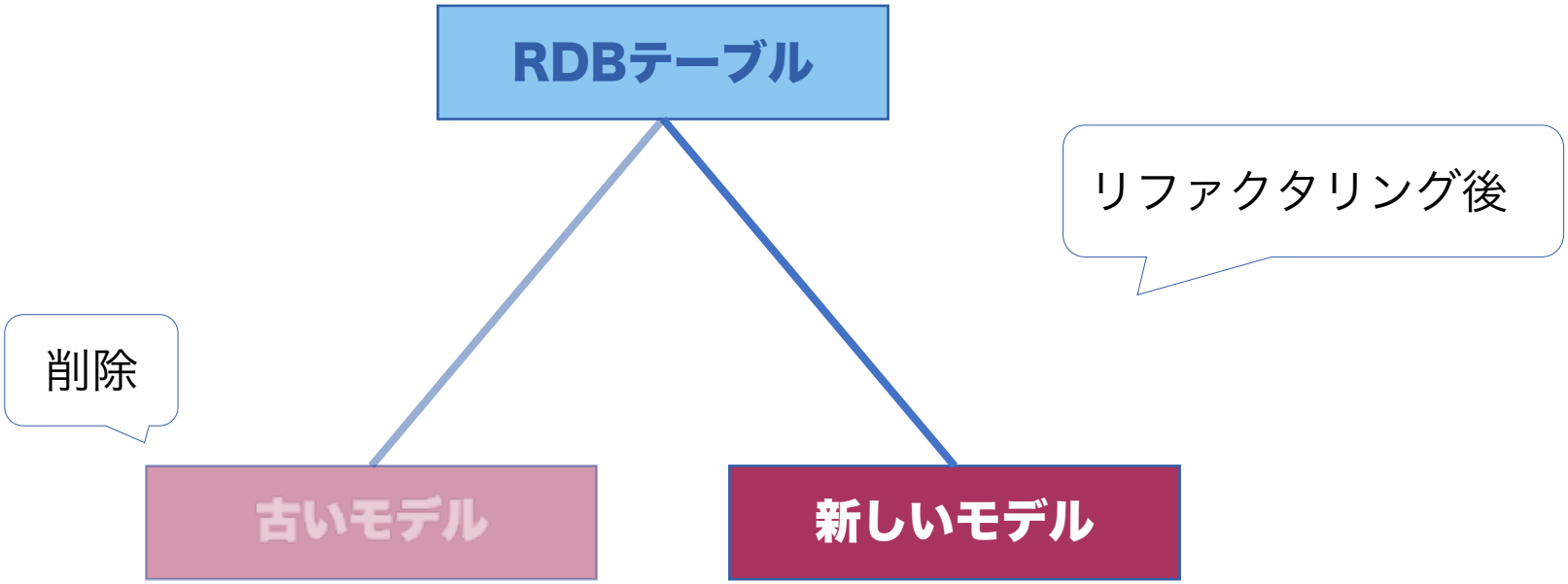
モデル

- 古いモデルと新しいモデルを共存させる



モデル

- 古いモデルと新しいモデルを共存させる



モデル

- 必要に応じてロジックをPOROに分離
  - Form, Finder, Decorator, Service

コントローラとビュー

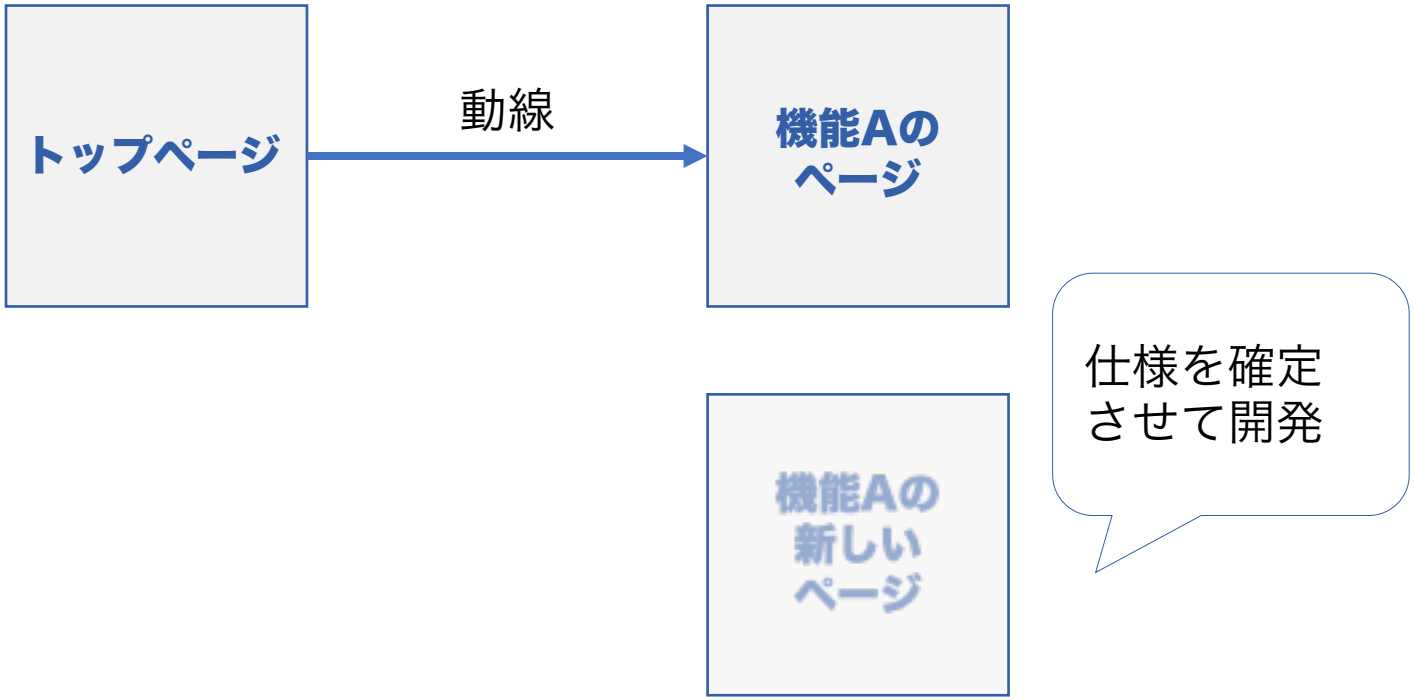
- 新しい機能を動線なしで作ってから動線を置き換える

コントローラとビュー

- 新しい機能を動線なしで作ってから動線を置き換える

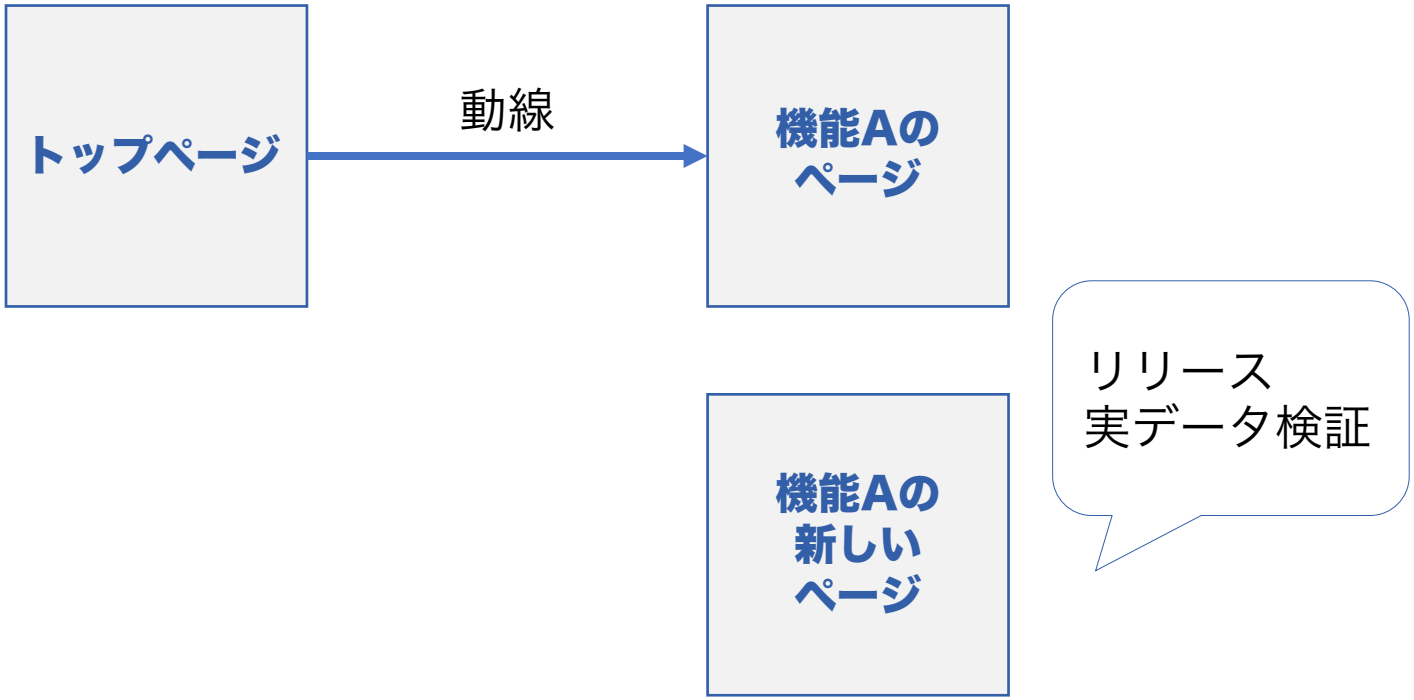
コントローラとビュー

- 新しい機能を動線なしで作ってから動線を置き換える



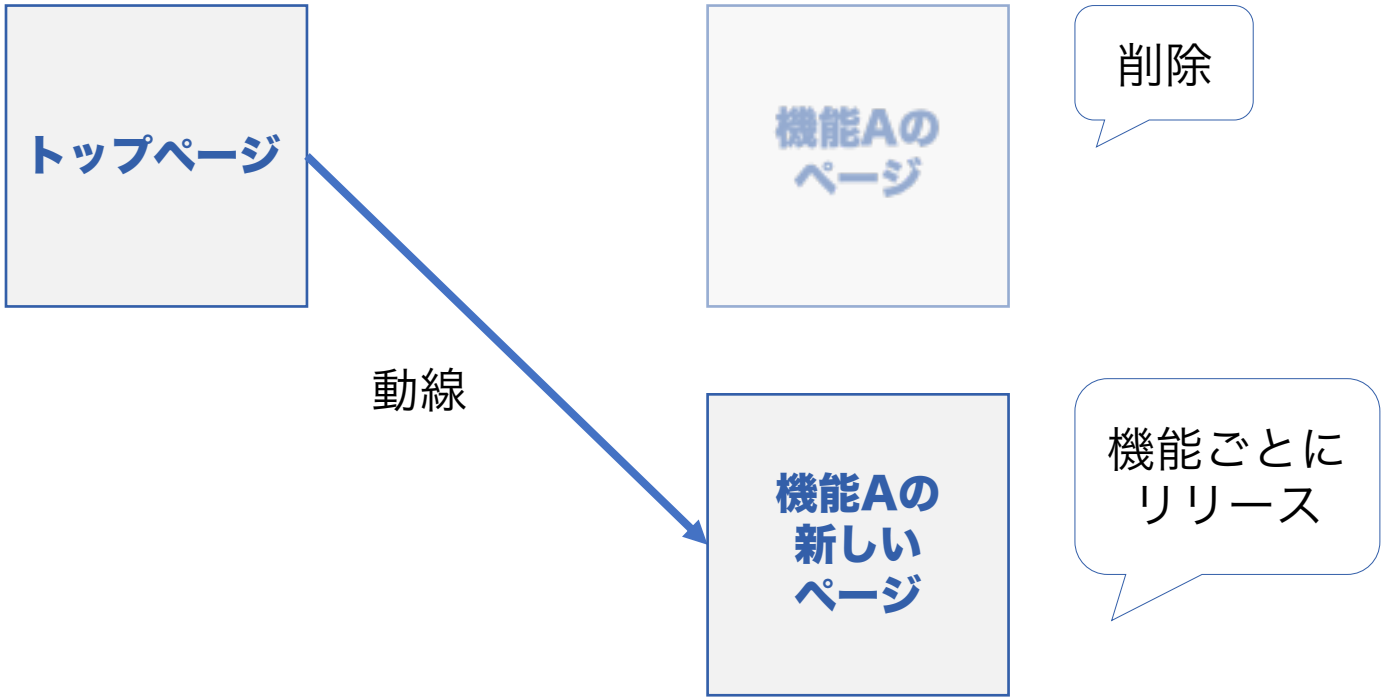
コントローラとビュー

- 新しい機能を動線なしで作ってから動線を置き換える



コントローラとビュー

- 新しい機能を動線なしで作ってから動線を置き換える



コントローラとビュー

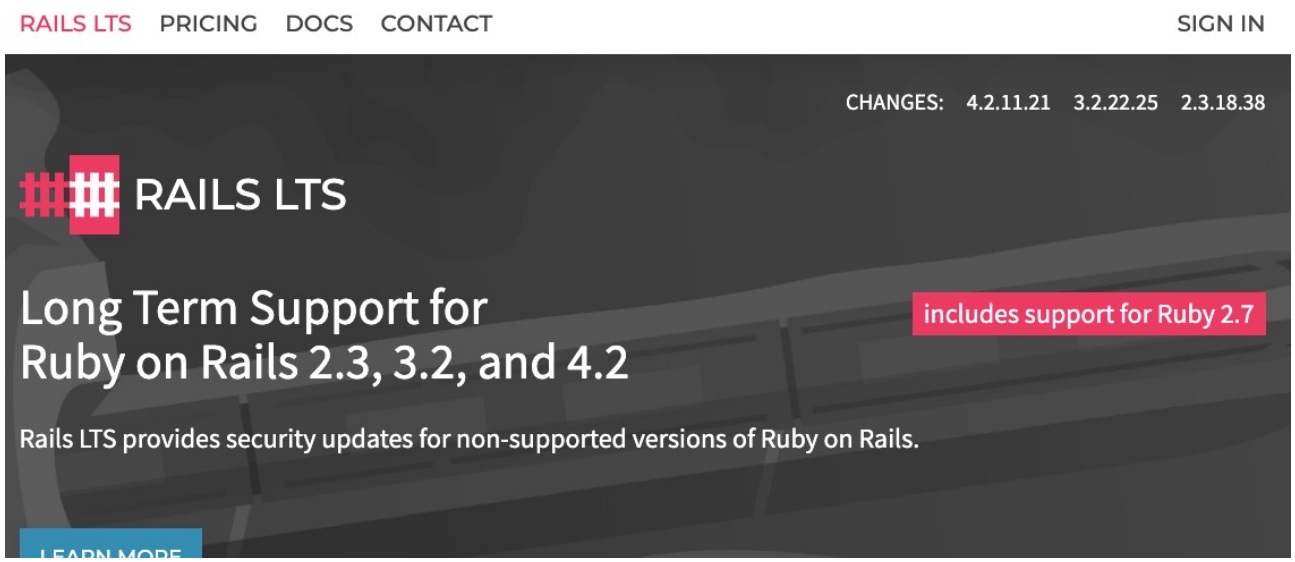
- 認可gemとしてbankenを導入
- コントローラにもビューにもロジックを書かない

こうして、リファクタリングPJが始まる

でもひとつだけ、看過できない問題が

セキュリティ問題

Rails LTS



※引用: https://railslts.com/ (2021/10/10閲覧)

ちなみに

- PJ開始の半年後くらいに、発表者が入社
  - このとき、リファクタリング前のコードを実際に見ています

PJ開始から3年後の2020年2月、
ついにリファクタリング完遂

不要になったコードの削除

219 files +45 -14337

497 files +178 -61248

	before	after
コントローラ数	157	200
アクション数	1,890	850
モデル数	184	166
Rubyの行数	97,000	45,000
ERBの行数	150,000	45,000
テストカバレッジ（行数）	-（1,500）	95%（120,000）

	before	after
コントローラ数	157	200
アクション数	1,890	850
モデル数	184	166
Rubyの行数	97,000	45,000
ERBの行数	150,000	45,000
テストカバレッジ（行数）	-（1,500）	95%（120,000）

いよいよRailsアップグレード

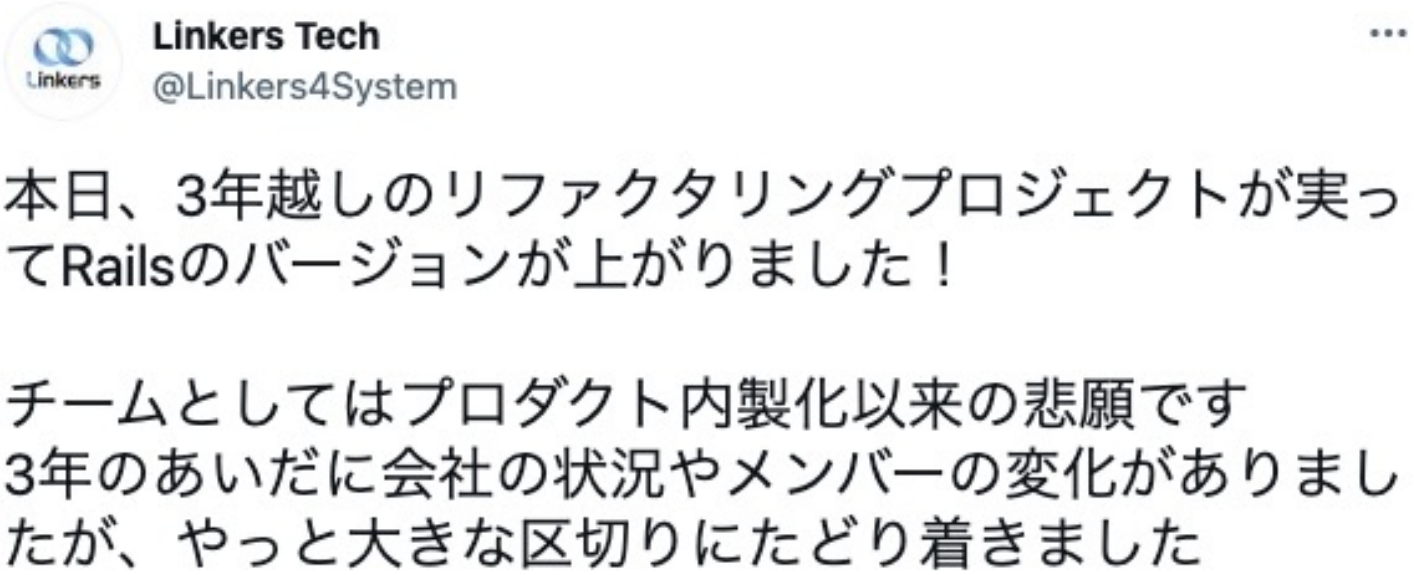
アップグレード先がサポート切れという現実

- 普通はgemを先に上げてRailsを少しずつ上げる
- しかし、4系はすでにサポート切れ
  - gemだけ上げると動かないケースも
- Railsと他のgemを一度にバージョンアップ
  - 動かない箇所だけ直す

テストが通るまで修正

テストが通ったらビジネスサイドを巻き込んで動作確認

2020年5月、とうとうリリース



※紆余曲折あって、Rails4でリリースせず一気にRails5まで上げました

2020年7月、ついに……

Linkers Tech
@Linkers4System

たった今、Linkers.net のRailsのバージョンが5.2から6.0に上がりました🎉

3ヶ月前の5/18までRails3だったサービスがとうとう最新版の6までアップグレードされ、3年前のリファクタリングから始まったプロジェクトがついにゴールを迎えました！

Railsバージョンが倍になった🎉

アップグレードして改善したところ

- セキュリティパッチをすぐ当てられるようになった
- 公式マニュアルが利用しやすくなった
- パフォーマンスの向上（体感）

めでたしめでたし

おわり

うそです🤗

- まだ解決していない問題
  - 今後、リンカーズのシステムをどう運用していけば、技術的負債が積み上がらなくなる？

2021/12/16 RubyWorld Conference 2021 | Shu OGAWARA (@expajp)

Railsバージョンを倍にしたサービスのそれまでとそれから

第1部

「それまで」

完

第2部 「それから」

技術的負債を解消しつづけるチームへ

アップグレードの少し前の出来事①

- CTO: Sさんが退職
  - 発表者がリンカーズ開発のリードをすることに

サービスに責任を持つ立場になった

今後もきれいなコードのままで
開発を続けられるとは限らない

なんで技術的負債が積み上がって
しまったんだろう？
そもそも技術的負債ってなんだ？

技術的負債

最初のコードを出荷することは、
借金をするようなものだ。
少々の借金は、書き換えによって
速やかに返済される限り、開発を加速させる。
危険なのは、その負債が返済されないときだ。

- Ward Cunningham(1992)

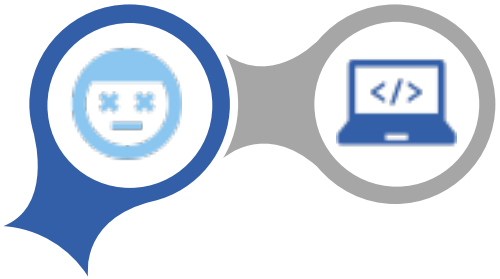
※翻訳の引用: https://ja.wikipedia.org/wiki/技術的負債 , 原文: http://c2.com/doc/oopsla92.html
いずれも2021/10/15閲覧

技術的負債＝
ソフトウェアの変更可能性を妨げるもの

ではソフトウェアに変更可能性が必要なのはなぜ？

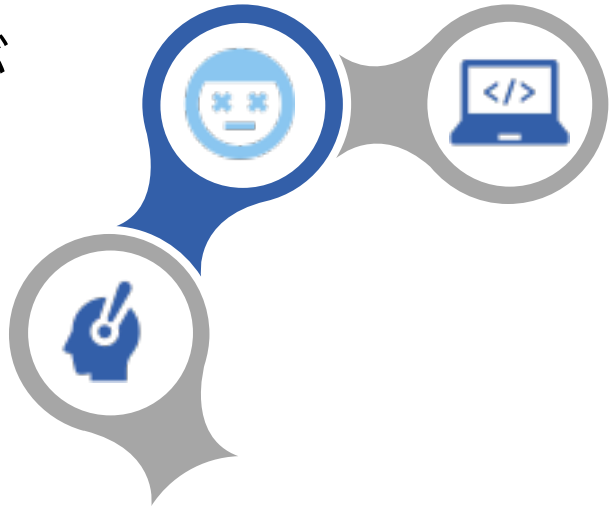
ソフトウェアが
変更される

ソフトウェアが
使いにくくなる



ソフトウェアが
変更される

ソフトウェアが
使いにくくなる



ソフトウェアが
変更される

ビジネスモデルを
変更する

ソフトウェアが
使いにくくなる

ビジネスモデルを
変更する

ビジネスモデルが
陳腐化する

ソフトウェアが
変更される

ソフトウェアが
使いにくくなる

ソフトウェアが
変更される

ビジネスモデルを
変更する

ビジネスモデルが
陳腐化する

ビジネス環境が
変化する

ソフトウェアが
使いにくくなる

ソフトウェアが
変更される

ビジネスモデルを
変更する

変更が市場に
影響を与える

ビジネスモデルが
陳腐化する

ビジネス環境が
変化する

社会情勢や法律、
社内の変化も含む

ソフトウェアには常時変更が必要

ソフトウェアが
使いにくくなる

ソフトウェアが
変更される

ビジネスモデルを
変更する

変更が市場に
影響を与える

ビジネスモデルが
陳腐化する

ビジネス環境が
変化する

社会情勢や法律、
社内の変化も含む

ソフトウェアとビジネスの変化で生まれるもの

- 不要になった機能
  - ビジネスが変化する以上は必ず生まれる
- 考慮不足のコード
  - 人が書く以上は必ず生まれる
  - すべての変更を予想したコードは書けない

いずれも、
コードの変更を
難しくする

つまり、技術的負債には2種類ある

第1部では別に
扱っていたが、
これも技術的負債

1 機能面での技術的負債（機能負債）

2 コード面での技術的負債（コード負債）

常時、ソフトウェアが変更されることで
技術的負債が溜まり続けるなら、
常時、解消し続けなくてはならない

これを踏まえて

改めて技術的負債が積み上がった理由をふりかえる

1 **創業時に エンジニア不在**

システムの技術的品質をチェックできなかった

2 **創業期は開発優先**

最低限の安全性を担保して、PMF達成を目標に

3 **交通整理が必要になった開発要望**

人が増えると要望の反映にも体制が必要に

改めて技術的負債が積み上がった理由をふりかえる

解決

社内には
エンジニアがいる

1 **創業時に エンジニア不在**

システムの技術的
品質をチェック
できなかった

2 **創業期は開発優先**

最低限の安全性
を担保して、
PMF達成を目標に

3 **交通整理が必要 になった開発要望**

人が増えると
要望の反映にも
体制が必要に

改めて技術的負債が積み上がった理由をふりかえる

解決

社内には
エンジニアがいる

半分未解決

激しい変更は
減ったが開発する
内容は多い

1 **創業時に
エンジニア不在**

システムの技術的
品質をチェック
できなかった

2 **創業期は開発優先**

最低限の安全性
を担保して、
PMF達成を目標に

3 **交通整理が必要
になった開発要望**

人が増えると
要望の反映にも
体制が必要に

改めて技術的負債が積み上がった理由をふりかえる

解決

社内には
エンジニアがいる

半分未解決

激しい変更は
減ったが開発する
内容は多い

未解決

交通整理する人は
まだいない

1 **創業時に
エンジニア不在**

システムの技術的
品質をチェック
できなかった

2 **創業期は開発優先**

最低限の安全性
を担保して、
PMF達成を目標に

3 **交通整理が必要
になった開発要望**

人が増えると
要望の反映にも
体制が必要に

改めて技術的負債が積み上がった理由をふりかえる

解決

社内には
エンジニアがいる

半分未解決

激しい変更は
減ったが開発する
内容は多い

未解決

交通整理する人は
まだいない

1 **創業時に
エンジニア不在**

システムの技術的
品質をチェック
できなかった

コード負債が溜まる

3 **交通整理が必要
になった開発要望**

人が増えると
要望の反映にも
体制が必要に

改めて技術的負債が積み上がった理由をふりかえる

解決

社内には
エンジニアがいる

半分未解決

激しい変更は
減ったが開発する
内容は多い

未解決

交通整理する人は
まだいない

1

**創業時に
エンジニア不在**

システムの技術的
品質をチェック
できなかった

コード負債が溜まる

機能負債が溜まる

余分な機能負債を作る

やるべきこと

機能負債

コード負債

これらを生まれるのを防ぎ、解消しつづけるしくみを考える

機能負債のためのしくみ

チームリーダーになったときはこの体制だった

- 直接チャットで問い合わせが来る

チャット張り付いてないといけないし、
チャット来るたび作業止まるし、
人が相手だから無下にもできない……

木こりのジレンマに陥る

- 問い合わせ対応に忙しくて、改善の時間が取れなかった

エスカレーションした結果、
プロダクトバックログを作ってもらった

プロダクトバックログ (PBL)

プロダクトバックログは、
創発的かつ順番に並べられた、
プロダクトの改善に必要なものの一覧である。

- Scrum Guide 2020

※引用: https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-Japanese.pdf
2021/10/16閲覧

アップグレードの少し前の出来事②

- もうひとりのSさんが入社
  - 機能と情報のとりまとめに着手していた

この人がプロダクトオーナー(PO)に就任

- PBLを精査し、優先順位をつける人
  - 起票するときはこの人を通す
- 機能の棚卸がされるようになった
- 発表者が木こりのジレンマから解放

コード負債のためのしくみ

コード負債を解消するときのアンチパターン①

- 気付いたら直す
- 手の空いたときに直す

コード負債を解消するときのアンチパターン①

- 気付いたら直す

絶対後回しにして忘れる

- 手の空いたときに直す

コード負債を解消するときのアンチパターン①

- 気付いたら直す
- 手の空いたときに直す

絶対後回しにして忘れる

そんなときは来ない

コード負債を解消するときのアンチパターン②

- 気付いたら起票してまとめて直す

コード負債を解消するときのアンチパターン②

- 気付いたら起票してまとめて直す

気付いたときに
書き残して作業に戻れる
のは良い

時間取って
まとめて直せるのも良い

コード負債を解消するときのアンチパターン②

- 気付いたら起票してまとめて直す

好き勝手に起票すると
本人しか知らない
チケット大量発生

起票者がやることに

「言い出しっぺの法則」
になるので、
誰も起票しなくなる

コード負債の解消に必要なしくみ

1. 気付いたときに書き残せる
2. 定期的に時間確保できる
3. 解消したい技術的負債への問題意識を共有できる
4. 言い出しっぺの法則が適用されない

「整地タスク」というしくみを考えた

整地タスクのしくみ

①普段の開発中に記入

技術的負債に気付いたら
専用スプレッドシートに
記入

1日/1週/それ以上
で工数を記入

この時点では
チケットを起票しない

整地タスクのしくみ

①普段の開発中に記入

技術的負債に気付いたら
専用スプレッドシートに
記入

1日/1週/それ以上
で工数を記入

②定期的に全員で確認

起票者が内容を説明

タスクごとに投票して、
優先順位付けしたり
却下したりする

全員で問題意識を共有
雑すぎる内容をリジェクト

整地タスクのしくみ

①普段の開発中に記入

技術的負債に気付いたら
専用スプレッドシートに
記入

1日/1週/それ以上
で工数を記入

②定期的に全員で確認

起票者が内容を説明

タスクごとに投票して、
優先順位付けしたり
却下したりする

③「1日」は月1の整地日で取り組む

全員が整地タスクをやる
「整地日」を設定

優先順位付けされた
バックログの上から取る

整地タスクのしくみ

①普段の開発中に記入

技術的負債に気付いたら
専用スプレッドシートに
記入

1日/1週/それ以上
で工数を記入

②定期的に全員で確認

起票者が内容を説明

タスクごとに投票して、
優先順位付けしたり
却下したりする

③「1日」は月1の整地日で取り組む

全員が整地タスクをやる
「整地日」を設定

優先順位付けされた
バックログの上から取る

④「1週」はスプリントに混ぜ込む

スプリントに1個だけ
整地タスクを混ぜ込む

通常の開発タスク同様に
見積して実装

両方の負債を解消するためのしくみができた

機能負債

コード負債

みたび技術的負債が積み上がった理由をふりかえる

解決

社内には
エンジニアがいる

半分未解決

激しい変更は
減ったが開発する
内容は多い

未解決

交通整理する人は
まだいない

1 **創業時に
エンジニア不在**

システムの技術的
品質をチェック
できなかった

2 **創業期は開発優先**

最低限の安全性
を担保して、
PMF達成を目標に

3 **交通整理が必要
になった開発要望**

人が増えると
要望の反映にも
体制が必要に

みたび技術的負債が積み上がった理由をふりかえる

解決

社内には
エンジニアがいる

半分未解決

激しい変更は
減ったが開発する
内容は多い

未解決

交通整理する人は
まだいない

1

**創業時に
エンジニア不在**

システムの技術的
品質をチェック
できなかった

コード負債が溜まる

機能負債が溜まる

余分な機能負債を作る

みたび技術的負債が積み上がった理由をふりかえる

解決

社内には
エンジニアがいる

解決

「整地タスク」で
開発を続けつつ
技術的負債を解消

未解決

交通整理する人は
まだいない

1 **創業時に
エンジニア不在**

システムの技術的
品質をチェック
できなかった

コード負債が溜まる

機能負債が溜まる

余分な機能負債を作る

みたび技術的負債が積み上がった理由をふりかえる

解決

社内には
エンジニアがいる

解決

「整地タスク」で
開発を続けつつ
技術的負債を解消

解決

POが要望を取り
まとめ、PBLで
優先順位付け

1 **創業時に
エンジニア不在**

システムの技術的
品質をチェック
できなかった

コード負債が溜まる

機能負債が溜まる

余分な機能負債を作る

結局、最も大事だと思ったこと

ビジネスサイドとのコミュニケーション

ソフトウェアが
使いにくくなる

ビジネスモデルを
変更する

ビジネスモデルが
陳腐化する

ソフトウェアが
変更される

変更が市場に
影響を与える

ビジネス環境が
変化する

社会情勢や法律、
社内の変化も含む

ビジネスサイドとのコミュニケーション

- ビジネスとシステムのズレを
  少なくすることで、
  機能負債の蓄積を遅らせる
- 同期的コミュニケーションが大事
  - モヤモヤを言葉にするきっかけ
  - 使う人と作る人の理解の分断を防ぐ

いまでは、最新の
Rails 6.1.4.1です

こうしてリンカーズは、
Railsアップグレードを果たしたあとも
技術的負債を解消しながら
運用されるサービスとなった

第1部のまとめ

「それまで」

- やむを得ない事情で
  技術的負債が積み上がった
- 運用しながらリファクタリング
  - 機能ごとに仕様を確定
  - MVCぜんぶ遷宮
  - 置き換わらなかったコードを削除

第2部のまとめ

「それから」

- 技術的負債には
  機能負債とコード負債がある
- 技術的負債は溜まりつづける
- 大事なのは、
  解消しつづけるしくみを作ること

さいごに

- 奇跡的な要素がかみ合った結果なので、
  全体での再現性は非常に低いと思います
- しかし、個々には再現性が持たせられるはずです
  - MVCぜんぶ遷宮
  - 画面ごとに仕様を確定させる
  - プロダクトバックログの設置
  - 整地タスクの運用

この発表が、
技術的負債に苦しむ
すべての人々への
エールになればと思っています

ところで、

「SaaS版リンカーズ」の話を覚えていますか？

We’re Hiring!!!

- 当時、実は新規開発となったシステム「LFB」が現在急成長中です
  - 全国の金融機関やグループ企業の顧客どうしを結びつけ、日本経済を活性化させるためのシステムです

We’re Hiring!!!

- 一緒に働く仲間を探しています！
  - 技術顧問には @yukihiro_matz さん、@netwillnet さんのお二人に就いていただいています
  - 興味のある方は @expajp まで

謝辞

- 退職済みにもかかわらず、インタビューに応じてくださった元CTO: Sさん
- 当時の話を聞かせて下さった元リファクタリングPJ専任のIさん
- この話を発信するため奔走してくれた上司
- この話を発信することをOKしてくれた経営陣
- レビューして下さった技術顧問のお二人と同僚たち、そして友人たち
- この発表をRWCに持っていくことをすすめてくださった、技術顧問のまつもとさん
- 動画を編集してくださったデザイナーのNさん

本当にありがとうございました

素材

- human pictogram 2.0
  - https://pictogram2.com/
- ICOOON MONO
  - https://icooon-mono.com/
- 24Slides
  - https://24slides.com/
- Wikimedia Commons
  - https://commons.wikimedia.org/