リファクタリング 目的・パターン・思考

Repro inc. @joker1007

はじめに

この話におけるリファクタリングの定義について

ユーザーの体験を変えずにコードベースを改善すること

として話をします。

長く続いたソフトウェアは複雑化する

特に一度顧客が付くと簡単に止められなくなる
機能や動きを維持しつつ開発しなければならない

複雑さも相俟ってどんどん開発が遅くなる

開発速度が遅くなる要因

- ある機能に影響するコード量が多く理解しきれない
  - 自信をもって変更できない
  - テスト範囲が膨大になり時間がかかる
- 機能同士が密結合している
  - ある機能を改修する時に芋蔓式に修正箇所が増大する

リファクタリングが目指すもの

- 読み易さの向上
- 疎結合化
- 不要なコード(複雑さ)の削除

これらを達成することで開発速度を回復させ、キャッチアップ速度の向上やバグの低減を目指す。

危険シグナル

- 読み辛くて訳が分からん
- なんでこのメソッドがここにあるんだ？
- 他のオブジェクトのメソッド呼び過ぎ
- このメタプロ必要か？
- やりたい事に比してこんな複雑な訳がない
- 作業見積りから乖離があり過ぎ
- なんかイラつく (最後は直感)

いつやるのか

- イラついた時
  - すぐ終わりそうなら
  - もしくは我慢ならなくなったら
- フィーチャ開発時に作業見積りに混ぜる
  - 締切がシビアだと厳しい
  - 見積り時にヤバそうな箇所を知っておく必要がある
- まとめて注力する期間を用意
  - 経営判断が必要
  - でかくて明確な目的とマッチする
- 破綻寸前
  - 時間の説得はしやすいが、ほぼ手遅れ

息を吸う様に日常に入ってるのが理想。

リファクタリング手法

ここからはリファクタリングとして、普段どういうことをやっているかを紹介していきます。

メソッド分割

一つの長大なメソッドを段落ごとに名前を付けて別メソッドに抽出する

効果:

- 読み易さの向上
- モックポイントの挿入

ただし、本質的な複雑さには変化が無い。
処理の粒度を揃えることが重要。リーダブルコードを読むべし。

完全コンストラクタ

不要なパラメータ渡しの無駄を避けて、オブジェクト生成時に用意できるものを全て揃える。

効果:

- オブジェクトを不変にしやすい
- 一度作ったら呼び出し元がオブジェクトの細かい挙動を知らなくて良くなる

シンプルなことだが、既存のコードに乗って書いてると気付かない事もしばしば。
次の「責任範囲の適正化」と連携しやすい。

責任範囲の適正化

デメテルの法則や、尋ねるな命じろ、の原則に反するコードを適切な場所に配置しなおす。
大体のケースでは、呼び出し元から呼び出し先にロジックを移動することになる。

効果:

- 不要なパブリックメソッドの削減
- テスタビリティ向上

メソッドチェインが減って、あるオブジェクトが元々知っている情報だけで処理が進む様になる。
トランザクション管理と実際の処理の境界を意識すると見通し良くなる場合がある。

クラス分割

責任が多過ぎるクラスをサブクラスに分割する。
オブジェクトの生成が複雑過ぎるなら専用のクラスを用意するなど。

効果:

- 名付けができる
- テスタビリティの向上
- 機能追加の際の影響範囲の限定

Rubyにはパッケージスコープの仕組みが無いので、作ったサブクラスが他の箇所から使われない様にコメントに書く
か、`private_constant`にする等の注意をすること。

パラメータオブジェクト

検索条件等、パラメータが大量かつデフォルト値があったり無かったりする場合に、パラメータをハンドリングするStructを用意する。

効果:

- メインロジックから瑣末な値の調整を分離できる
- デフォルト値の扱いだけを簡単にテストできる

ストラテジーパターン/ステートオブジェクト

オブジェクトのタイプや状態によって、処理内容が変わる場合に処理を専用のオブジェクトに移譲する。

効果:

- 状況に応じて変化する処理の影響範囲が明確になる
- Cyclomatic complexityが減少する

Rubyなら動的ディスパッチと規約で同様のことができるが、ちゃんとクラスを作った方が呼び出しが明確になるし安全になる。

gem化

汎用化できそうな処理をgemとして分離する。社内でのredisの扱い方とか、認証パターンとか、AWSのAPIクライアントラッパーなど。

効果:

- アプリケーション本体と独自に変更可能
- テスト境界が自明になるので、網羅的なテストが書き易い
- OSS化のチャンス

汎用化の分設計コストがかかる。単機能のgemにしておかないと逆に大変になる可能性がある。

ミドルウェアに頼る

fluentd等のミドルウェアとクライアントライブラリやラッパーライブラリを組み合わせて機能を代替する。

効果:

- gem化と同様に責任を外部システムに移せる
- パフォーマンス向上に繋がる

メンテ対象が別途増える。運用コストとのバランスを取る必要がある。

DBスキーマ修正

DBの制約やデータの持ち方を修正することで、余計なvalidationやデータの引き直しを削減する。

効果:

- データ整合性が保証され、変更に対して安全になる
- パフォーマンス向上
- ActiveRecordの表現力向上

データが増えてくるとマイグレーションが死ぬ程大変になる。
しかし元が狂ってる場合、歪みがコード側にあることが多いので、どこかで着手する必要がある。

仕様変更

ユーザの体験を損ねない範囲で制約を緩和する様に仕様変更し、コードを簡易化する。
結果整合性で十分な場合に厳密なフロー制御を止めて処理を独立させる等。

効果:

- 根本から複雑さに対処できる
- パフォーマンス向上に繋がることも多い

開発チームだけでは話が終わらない。
話の早いProduct Managerが居ると楽。
提案するにも、ドメイン知識とユーザーに与えている価値の本質を知っておく必要がある。
しかし、効果はかなり大きい。

機能削除

価値の低い機能をそもそも廃止する。

効果:

- コード群を完全に消せる
- 最強

とにかく話を通すのが大変。営業方法や経営判断に影響する場合もある。
しかし数ヶ月分の工数に影響する場合もある。

実例

(会場限定)

リファクタリングに必要な思考

- コードの良し悪しについての指針を持つ
- 何が無くせれば、何ができるかを考える
- 既存のコードを信用するな
    - 全てのコードはもっと上手く書ける
- 仕様は絶対ではない

非常に大事なことは

顧客に提供している価値が何なのかを知ること

既存の機能が本当に必要なのかを常に考えること

より良い価値の提供のためには引き算も必要

まとめ

凝集度や責任範囲等コードの良し悪しにはちゃんと指針がある。
そして、改善内容にちゃんと説明を付けられないと駄目。

プログラミングだけでなくドメイン知識に無頓着では本質的な改善提案ができない。
仕様もコードも常に疑いを持って改善の余地を探しながら仕事していきましょう。