

僕の考えるユニットテストとのほどよい付き合い方

2022/02/02 TomoakiTANAKA

自己紹介

田中智章

- 富岡市出身
- SE としてキャリアを始め、現在は事業会社勤務
- 主に Web開発 を担当、時々アプリ開発 (Rails や Flutter)
- 好きなお酒はウィスキー全般、あと LEGO が好き



会社紹介

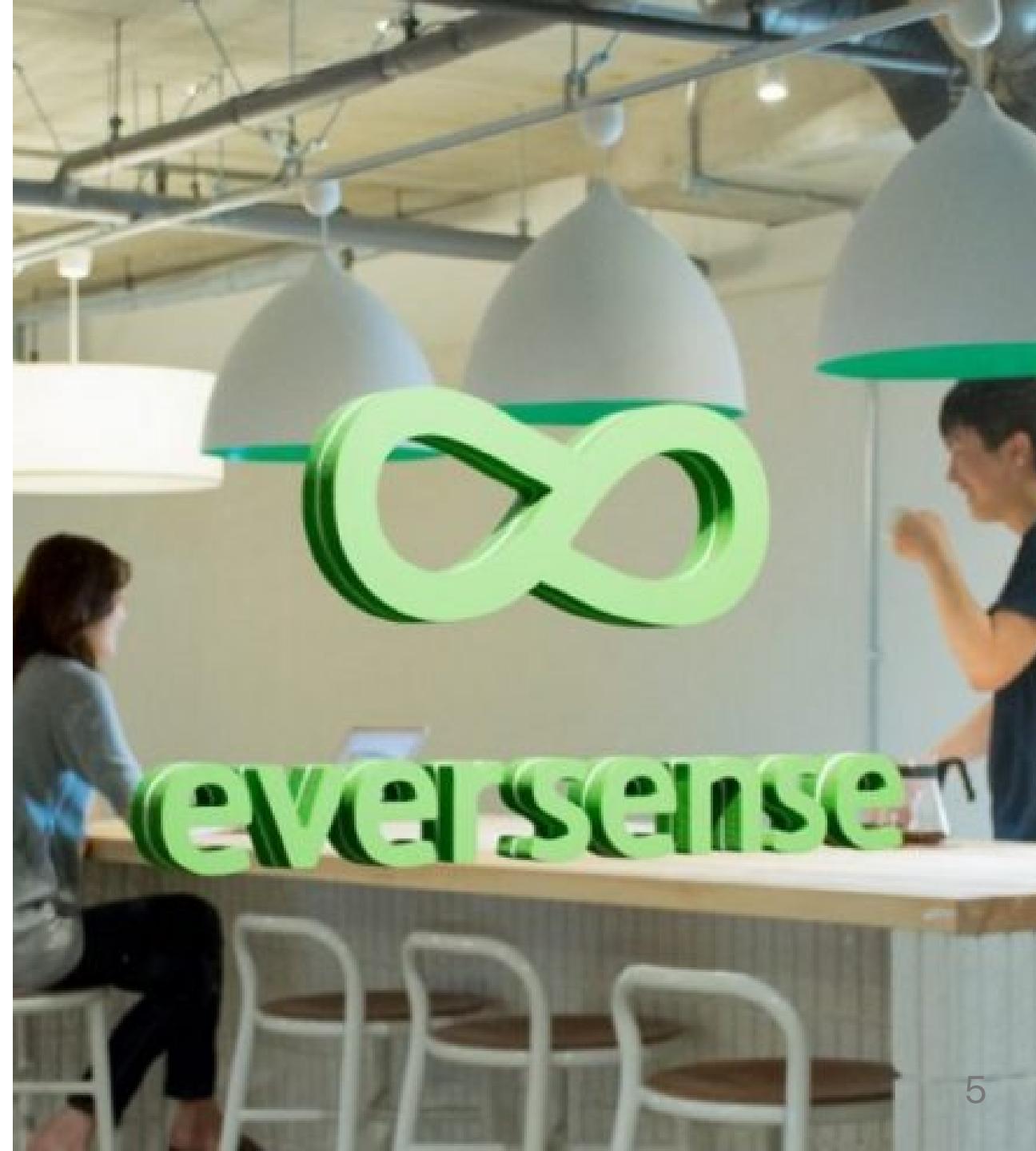
株式会社エバーセンス

Vision

- 家族を幸せにすることで、笑顔溢れる社会をつくる。

Products

- 妊婦さんへの情報提供をする
「ninaru」
- 子育てに必要な情報や機能満載の
育児アプリ 「ninaru baby」



アイスブレイク

- Q. みなさんは「ユニットテスト」と聞くとどういったイメージをお持ちでしょうか？
- Q. ユニットテストをなぜ行っていますか？
- Q. いつユニットテストを行いますか？

本日の勉強会について

本日の勉強会について

スピーカーのユニットテストに関する考え方や知見をお話します（経験談）。

僕（及び周辺）で実践している内容も紹介します。

明日からのユニットテストに何か活かせるものをお話できればと思っています。

対象

- 単体テスト書いていない人/書いている人
- どういうときにテストを書けばよいか指針がほしい人
- ユニットテストの運用方法の事例を知りたい人

目次

目次

1. スピーカとテストの歴史
2. ユニットテストとは？
3. いつ / どれだけユニットテストを作成し実行するか？
4. 事例紹介
5. まとめ

* 組織の意見でなく、個人の意見です

スピーカとテストの歴史

テストとわたし①

- 出会い
 - 高専のインターンシップ時
 - 書籍をベースに、境界値分析やホワイトボックス・ブラックボックステストを知る
- 大学次代（2010年頃～）
 - TDDやテストファーストの記事が流行っていた
 - xUnit系ライブラリの使い方や考えを知る

テストとわたし②

- 新卒時
 - 新卒なりに色を出そうとしてテストコードを書きはじめる
 - 大量のテストケース作成→動作チェックを経て、コードでやれる部分はコードでやったほうが効率的だなと実感
- 現在
 - テストは必要と判断したら書く
 - テストをかけるようにするとコード設計が洗練されるので、書かなくてもいいから書けるように思考するのが大事だなと思うこの頃

ユニットテストとは？

コンピュータプログラミングにおいて単体テスト（たんたいテスト）あるいはユニットテスト（英語: unit test）とは、ソースコードの個々のユニット、すなわち、1つ以上のコンピュータプログラムモジュールが使用に適しているかどうかを決定するために、関連する制御データ、使用手順、操作手順とともにテストする手法である

- 出典：[wikipedia](#)

???

僕の考えるユニットテスト

- あるメソッドに対して、期待する入力と出力を組み合わせを定義
- 組み合わせ通りの結果になることを保証する

```
it 'youtube動画へのリンク形式が不正なときに無効な状態であること' do
  # 入力の定義
  invalid_urls = %w[https://cojicaji.jp
                     https://youtube.com
                     https://www.youtube.com/@hoge
                     https://www.youtube.com/user/こじかじ
                     https://www.youtube.com/channer/こじかじ]

  # すべての入力に対して処理を行う
  invalid_urls.each do |invalid_url|
    user_with_sns.youtube = invalid_url
    user_with_sns.valid?

    # 入力と出力の組み合わせを検証
    expect(user_with_sns.errors[:youtube]).to include('リンクの形式が誤っています')
  end
end
```

いつ / どれだけユニットテストを作成し実行するか？

いつ、どうやって

- プルリクエスト作成時？リリース時？
- 手動？自動？

どこまで

- カバレッジは？C0、C1？
- 全メソッド？一部のメソッド？
- ロジックのみ？UI部分も？

ユニットテストする、しない

シンプルな問だが考慮する変数は多い

スピーカーの意見

いつ、どうやって

- プルリクエスト作成時にテストを用意しておき実行（手動）
- mainブランチへのマージ時にも実行（自動）

どこまで

- カバレッジは気にしない。後述する複雑系のみ対応
- 基本ロジックのみでUIテストはしない
- コード解析やリファクタリング目的で作成することも

いつ、どうやって、どこまで、に関する事例紹介

事例紹介

事例紹介（いつ、どうやって）

手動実行について

- ローカル環境を用意し「コマンド実行」できるようにしておく
- テストの実行方法を記載しておく（README、テストコード）
- テスト忘れが発生しないようチェックリスト（PULL_REQUEST_TEMPLATE.mdを利用）

自動実行について

- 簡易導入としてgit hooksを利用。push時にテスト
- 最近はGithub Actionsがメイン。無料枠が強力。
 - iOSアプリのビルドやストアへのアップロードにも利用している（余談）

例) 手順を記載しておく

```
# ↓↓↓ テストの実行方法を記述しておく
# run test command
# docker compose exec rails bin/rspec spec/models/user_spec.rb

require 'rails_helper'

RSpec.describe User, type: :model do
  # ...ユニットテストの中身...
end
```

例) Github Actions

```
name: dart-analyzer

on:
  # masterやfeatureブランチにpushした時
  push:
    branches: [ 'master', 'feature/*' ]
    paths: ['**.dart', 'pubspec.*', 'analysis_options.yaml', '.github/workflows/dart-analyzer.yaml']

jobs:
  # テストジョブを走らせる（コード解析もセット）
  test:
    runs-on: ubuntu-latest
    timeout-minutes: 5
    steps:
      - uses: actions/checkout@v2
      - uses: subosito/flutter-action@v1
        with:
          flutter-version: 2.2.0
          channel: 'stable'
      - name: run-analyze
        run:
          flutter pub get
          flutter test
          flutter analyze --fatal-warnings --no-fatal-infos
```

事例紹介（どこまで）

カバレッジ100%目標にしない

- 計算が直感的でない（例：日付処理）や、入力パターンが多く手動でのテストがめんどくさいものを優先的に行う
- 品質が担保できれば良いので、ほぼ自明なものをわざわざテストしない

ロジックに集中

- ロジックのテストに集中しUIのユニットテストはしない
(UIは見たほうが早いことも多く、テスト作成や維持するコストが重いので)
- ロジックに集中できるように、レイヤードアーキテクチャを意識。できる限りプレーンな言語だけでロジックを記載

例) 日付計算 (パターンは多岐に渡り、エッジケースも複雑なのでテスト化する)

```
// ロジック
int numberOfDayOfTheWeekInMonth() {
    // ある日付が7の周期で何回目の登場かを計算
    final nth = (_value.day - 1) / 7.floor() + 1;
    return nth;
}
```

```
// test
void main() {
    group('ある月における何回目の曜日か', () {
        test('テストデータが第1水曜日であること', () {
            final result = calendarDate.numberOfDayOfTheWeekInMonth();
            expect(result, 1);
            expect(calendarDate.dayOfWeek, DayOfWeek.Wednesday);
        });
    });
}
```

例) アプリのレビュー発火ロジック (起動100回を実施するのはめんどくさい。起動回数さえ正しければ～という考え方でロジックをテスト)

```
class ReviewLogic {  
    bool shoudFired() {  
        // 直接データを引っ張ってくると結合度が高くテストしにく  
        final getLaunchTime = getgetLaunchTimeFromDataBase();  
        return if getLaunchTime >= 100;  
    }  
}  
↓↓↓  
↓↓↓  
class ReviewLogic {  
    // リポジトリなどレイヤーをかませるとテストしやすい  
    final _repository = UserPropertyRepository();  
    bool shoudFired() {  
        final getLaunchTime = _repository.getgetLaunchTime();  
        return if getLaunchTime >= 100;  
    }  
}
```

まとめ

まとめ①（ユニットテストの考え方）

- 目的（品質担保）のするために、費用対効果の高い方法を取れば良い
 - 複雑な処理のエッジケースの確認
 - テスト自体がドキュメントになるので開発が持続するなら書く
 - など

まとめ②（ユニットテストする / しない）

する

- 入出力の組み合わせが多い場合
- コード解析やリファクタリングするときの前準備

しない

- 目視で十分な部分なところ
- 入力に対する出力がシンプルな場合（例：ユーザー名に「さん」をつける、だけ）
- 行数が大きいメソッド（テストの前に、メソッドを分解）

ご清聴ありがとうございました

付録

参考文献

- 影響を受けたり、役に立ったと思う書籍や資格
 - [テスト駆動開発](#)
 - ご存知t_wadaさんの翻訳本（KnetBeckの書籍としてもかなり有名）
 - [RSpecによるRailsテスト入門](#)
 - [JSTQB（ソフトウェアテストの資格）](#)
 - ソフトウェア開発していればおおよそ経験する内容を網羅的にまとめている
 - 1～3年目くらいにスキルの定着具合や抜け漏れないか確認する意味で

質疑応答

おわり