PHP Conference Japan 2024

テストコードのガイドライン
～作成から運用まで～

rikuto（@riku929hr）

自己紹介

rikuto（さとうりくと）

@riku929hr（X/github/mixi2 etc.）

2020卒

SWE@株式会社オープンロジ ← BizDev@ソフトバンク株式会社

一児（0歳）の父

野球観戦・楽器（ホルン）

Outline

ガイドライン作成を決意するまで

書籍から学んだテストの基本

ガイドライン作成のポイント

導入後の課題

まとめ・今後の課題

テストといってもいろいろありますが
ここでは開発者が行う自動テスト（単体テスト・統合テスト）を指します
以降、これらを単に「テスト」と呼ぶことにします

Outline

ガイドライン作成を決意するまで

書籍から学んだテストの基本

ガイドライン作成のポイント

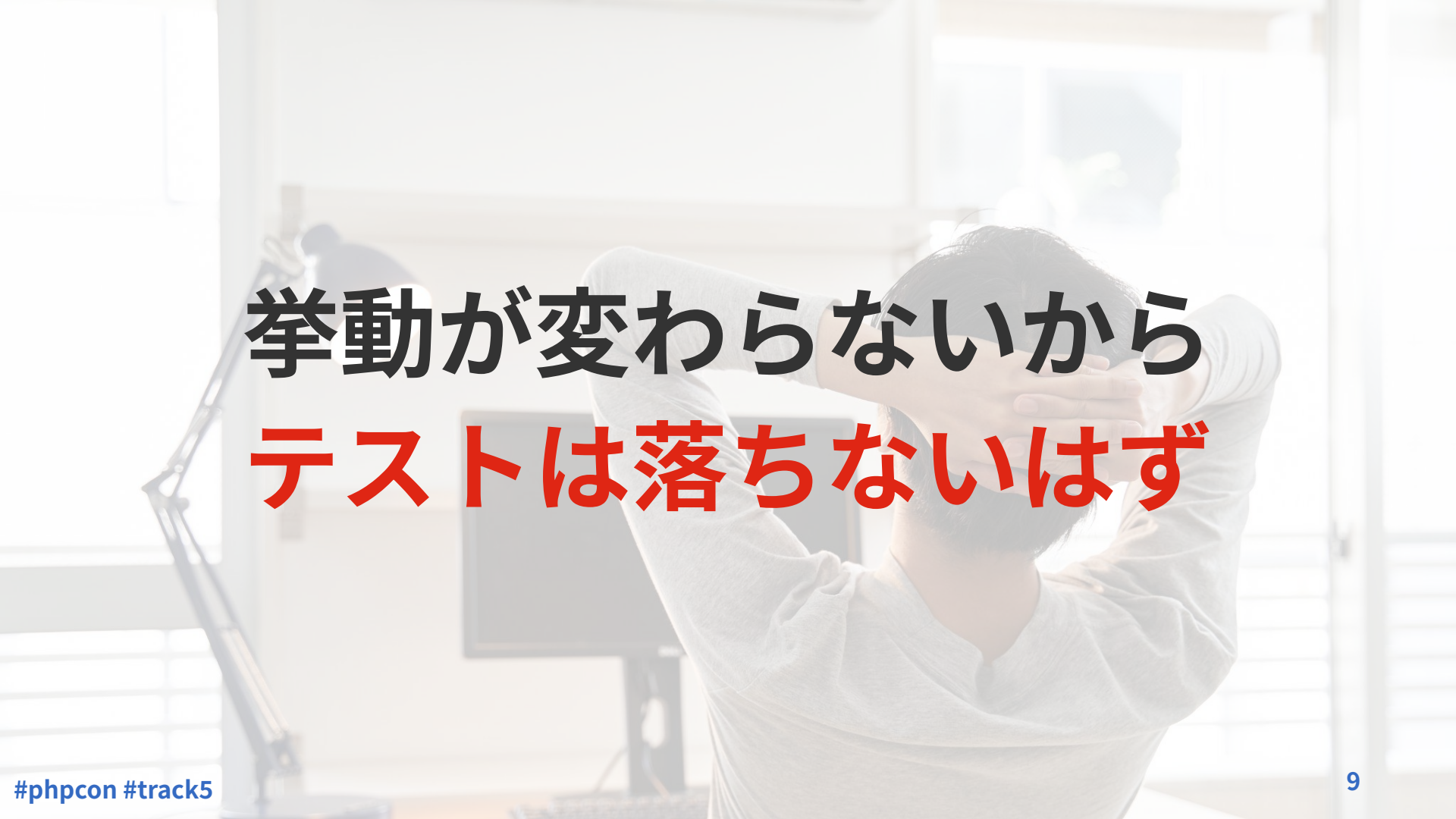
導入後の課題

まとめ・今後の課題

テストよくわからんエンジニアが
ガイドライン作成を
決意するまでのお話

とある開発タスクで
リファクタリングしてた

リファクタしたら
テストが落ちた！



挙動が変わらないから
テストは落ちないはず

よく見渡すとこんな状況

（すべてのコードではなく一部ですが）

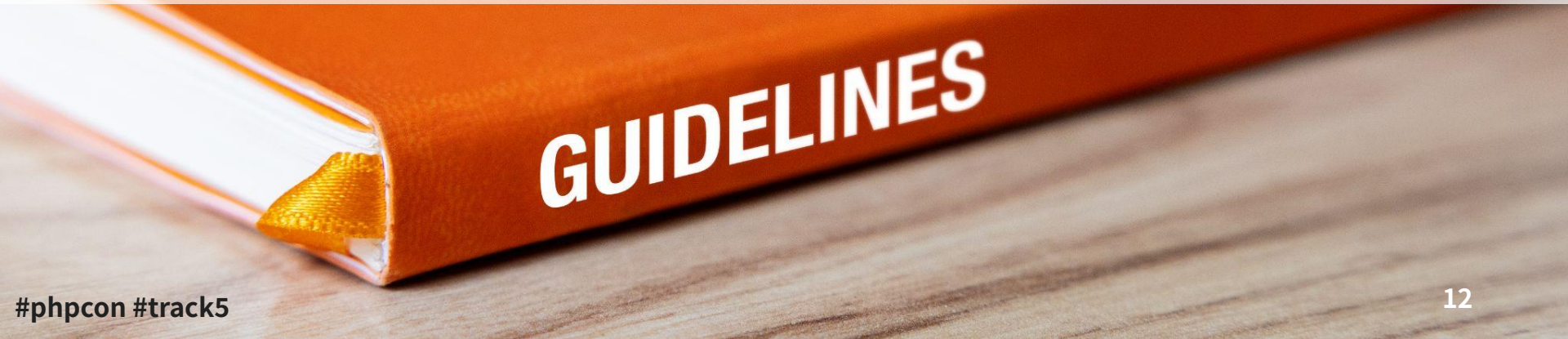
- **何を検証したいのか、パッと見でわからない**テストコードがある
- Controllerのテストだけ厚く書かれており、**実行に時間がかかってストレス**
  - オープンロジではLaravel（MVC）を使っています
- プロダクションコードはしっかりレビューされる一方、
  テストコードはほとんどレビューされないこともある

社内の機運の高まり

テストカバレッジが低く、テストに対する
問題意識が高くなっていた

開発組織として課題の共通認識はあったものの、
改善に着手できていない状態だった

ガイドライン作れば
いいのでは？



そもそもいいテスト・テストコードって何？

読書会やりました

「単体テストの考え方/使い方」

1回 1時間×15回 ＝ 15時間 かけて読みました

https://book.mynavi.jp/ec/products/detail/id=134252

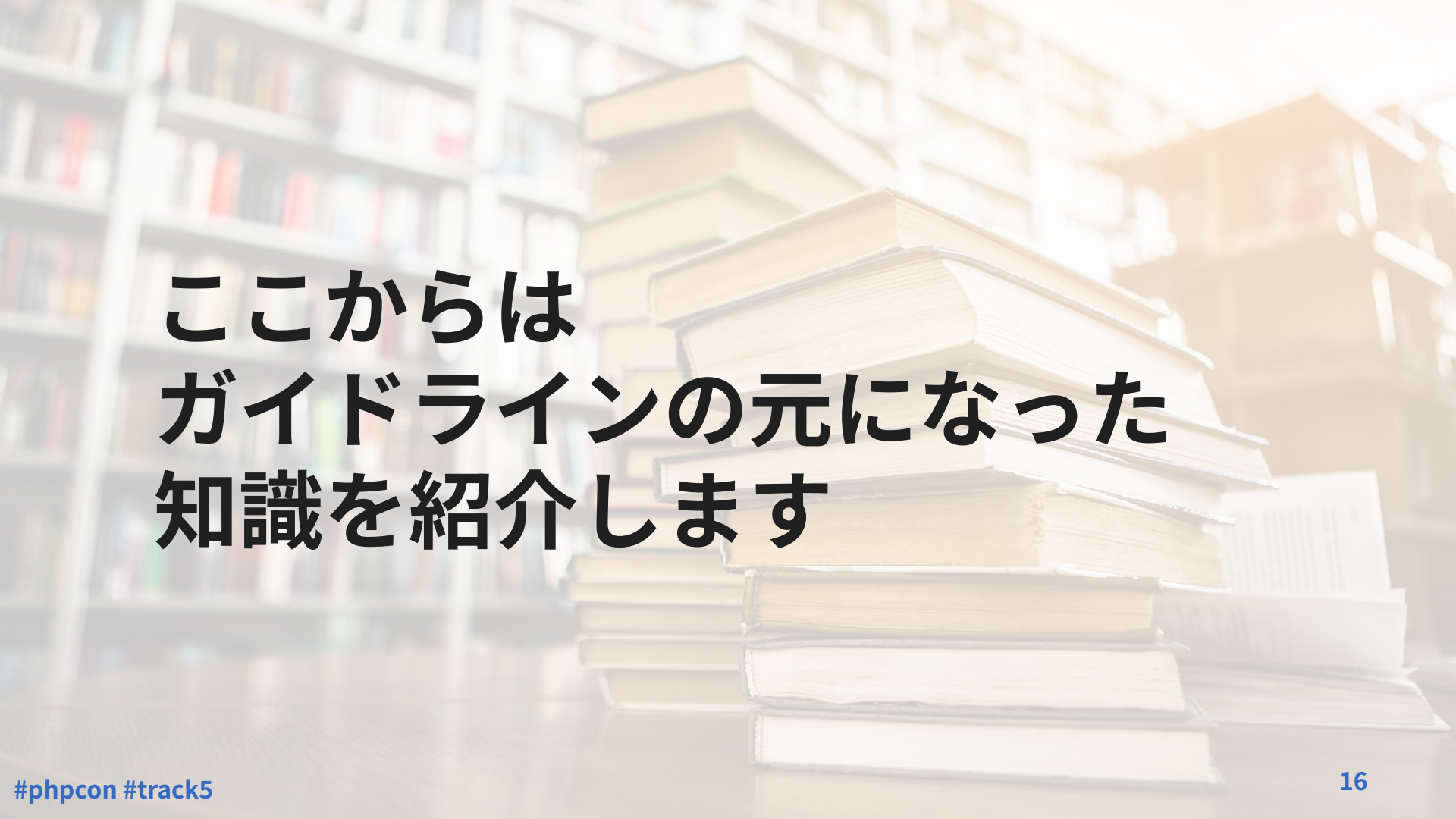
それ以外にお世話になった本たち

www.shuwasystem.co.jp/book/b620733.html

www.sbcr.jp/product/4815608750/

www.seshop.com/product/detail/25037

www.oreilly.co.jp/books/9784873119656/



ここからは
ガイドラインの元になった
知識を紹介します

Outline

ガイドライン作成を決意するまで

書籍から学んだテストの基本

- なぜテストするのか
- 振る舞いをテストするということ
- 何をどのくらいテストすればいいのか
- 認知負荷を下げるための工夫

ガイドライン作成のポイント

導入後の課題

まとめ・今後の課題

Outline

ガイドライン作成を決意するまで

書籍から学んだテストの基本

- **なぜテストするのか**
- 振る舞いをテストするということ
- 何をどのくらいテストすればいいのか
- 認知負荷を下げるための工夫

ガイドライン作成のポイント

導入後の課題

まとめ・今後の課題

なぜテストするのか

少なくともリグレッションテストにおける自動テストの目的とは、
成長し続けるプロダクトを**安全にリリースできる状態を、持続可能なコストにキープし続ける**ことだとわかります。（テスト自動化実践ガイド p29）

「**バグを捕捉すること**」は、テストの動機の一部にすぎない。
（中略）同等に重要なものは、**変化を可能とする能力を備えておく**ため、
という理由である。
新機能追加、コードの健全性に主眼を置いたリファクタリングの実施、
（中略）どれを行っていようが、**自動テストは間違いを素早く捕捉でき、そのおかげで信頼性を保ちつつソフトウェアを変更できる**ようになる。
（Googleのソフトウェアエンジニアリング 11章 テスト概観）

バグを早い段階で捕捉し、変更容易性を保つ

良いテストとは

- バグを正しく検出できる
- 自信を持ってコードを変更できる

良いテストとは

- バグを正しく検出できる
- 自信を持ってコードを変更できる

実装の詳細ではなく
振る舞いをテストすること

実装の詳細？？
振る舞い？？

Outline

ガイドライン作成を決意するまで

書籍から学んだテストの基本

- なぜテストするのか
- **振る舞いをテストするということ**
- 何をどのくらいテストすればいいのか
- 認知負荷を下げるための工夫

ガイドライン作成のポイント

導入後の課題

まとめ・今後の課題

実装の詳細と振る舞い

実装の詳細

具体的な処理の中身

（例）
テスト対象のコードから呼ばれる
関数が、想定通りに呼ばれている
ことを検証する

振る舞い

最終的な出力
関数が返す結果

（例）
テスト対象の処理の結果、
得られた値が意図したものに
なっていることを検証する

1つ例を挙げて
みてみましょう

実装の詳細のテスト（例）

代表例はモック[*]を利用したテスト

「1000円以上で100円割引」というロジックを考えてみる

※不完全なコードであることはご容赦ください

```
final class Checkout
{
    public function execute(int $totalPrice): int
    {
        if ($this->discountService->needsDiscount($totalPrice)) {
            // 割引処理
            $checkoutPrice = $this->discountService->discount($totalPrice);
        }
    }
}
```

この関数（execute）のテストについて discountServiceをモックにして書いてみる

[*] 厳密にはテストダブルという表現が正しいですが、説明を簡単にするため聞き馴染みのあるモックという言葉を使っています

実装の詳細のテスト（例）

```
public function test_1000円未満の場合は割引されない()
{
    $discountServiceMock = $this->createMock(DiscountService::class);
    $discountServiceMock
        ->expects($this->once())
        ->method('needsDiscount')  // needsDiscountメソッドが1度だけ
        ->with(500)                // 引数500で呼ばれることを確認して
        ->willReturn(false);       // falseを返すようにmockする

    $discountServiceMock
        ->expects($this->never()) // discountメソッドが呼ばれないことを確認
        ->method('discount');

    $checkout = new Checkout($discountServiceMock);
    $checkout->execute(500);
}
```

実装の詳細のテスト（例）

```
public function test_1000円未満の場合は割引されない()
{
    テスト対象のコード
    if ($this->discountService->needsDiscount($totalPrice)) {
        // 割引処理
        $checkoutPrice = $this->discountService->discount($totalPrice);
    }

    $discountServiceMock
        ->expects($this->never()) // discountメソッドが呼ばれないことを確認
        ->method('discount');

    $checkout = new Checkout($d
    $checkout->execute(500);
}
```

割引されない条件を
discountメソッドが呼ばれないことで
テストしている

仕様変更が入ったら…

1000円以上で100円の割引に加え、

無条件に50円引きが追加されたら…？

```
public function execute(int $total)
{
    if ($this->discountService->needsDiscount($total)) {
        $total = $this->discountService->discount($total);
    }
    // 50円引き！
    $checkoutPrice = $this->sale50YenOff($total);
}
```

discount()以外に割引処理が入った！

※雑なコードであることはご容赦ください

テストが仕様変更に追従しない

無条件に50円引きのはずだが…

```
public function test_1000円未満の場合は割引されない()
{
    $dis
    $dis

    $discountServiceMock
        ->expects($this->never()) // discountメソッドが呼ばれないことを確認
        ->method('discount');

    $checkout = new Checkout($discountServiceMock);
    $checkout->execute(500);
}
```

100円引きの関数（discount）の呼び出ししか確認していないので、本来失敗しなければいけないテストが通過してしまう

テストが仕様変更に追従しない

無条件に50円引きのはずだが…

```
public function test_1000円未満の場合は割引されない()
{
    100円引きの関数（discount）が呼ばれないことを確認していないので、
    本来失敗しなければいけないテストが通ってしまう

    $discountServiceMock
        ->expects($this->never()) // discountメソッドが呼ばれないことを確認
        ->method('discount');

    $checkout = new Checkout($discountServiceMock);
    $checkout->execute(500);
}
```

バグを正しく検出できない！

時間の都合上触れませんが
コードの変更もしづらくなります

どうすればいいのか

最終的な結果（＝振る舞い）を確認すれば、バグは正しく検出される

```
public function test_1000円未満では割引されない()
{
    $checkout = new Checkout();
    $this->assertSame(999, $checkout->execute(999));
}

public function test_1000円以上で100円割引される()
{
    $checkout = new Checkout();
    $this->assertSame(1400, $checkout->execute(1500));
}
```

関数を実行して得られた結果が、期待する値になっているかを確認する

振る舞いをテストすれば、テストも仕様変更に追従する

振る舞いをテストできるコードを書こう

実装の詳細を公開すると、振る舞いをテストしにくい

実装の詳細である
割引判定ロジックが
publicメソッド

```
final class DiscountService
{
    // 割引するかの判定
    public function needsDiscount(int $total): bool
    {
        return $total >= self::MINIMUM_DISCOUNT_PRICE;
    }

    // 割引処理
    public function discount(int $total): int
    {
        return $total - self::DISCOUNT_AMOUNT;
    }
}
```

割引判定と割引処理を
呼び出し先に委ねている

振る舞いをテストできるコードを書こう

振る舞いだけを公開すると、振る舞いのテストが書ける！

実装の詳細である
割引判定ロジックが
privateメソッド

```
final class DiscountService
{
    // 割引するかの判定
    private function needsDiscount(int $total): bool
    {
        return $total >= s
    }

    // 割引処理
    public function execute(int $total): int
    {
        // 割引判定
        if ($this->needsDiscount($total)) {
            return $total - self::DISCOUNT_AMOUNT;
        }
        return $total;
    }
```

振る舞いである割引処理の結果だけ
publicにして公開する

テックブログ書いてます

今回詳しく説明しきれなかった
実装の詳細のテストから振る舞いのテストへの
リファクタリングにも触れています

https://zenn.dev/openlogi/articles/unit-test-principles

モック[*]の使い方

モックの利用は最小限にするのがお得！

使ってはいけない、ということではない
テストしにくいものをテストするのには有益な道具

自分たちで管理しないリソース（管理下にない依存）だけにする

外部サービスのAPI

メール送受信

外部の通知システム

[*] 厳密にはテストダブルという表現が正しいですが、説明を簡単にするため
聞き馴染みのあるモックという言葉を使っています

Outline

ガイドライン作成を決意するまで

書籍から学んだテストの基本

- なぜテストするのか
- 振る舞いをテストするということ
- **何をどのくらいテストすればいいのか**
- 認知負荷を下げるための工夫

ガイドライン作成のポイント

導入後の課題

まとめ・今後の課題

課題として認識されていたこと

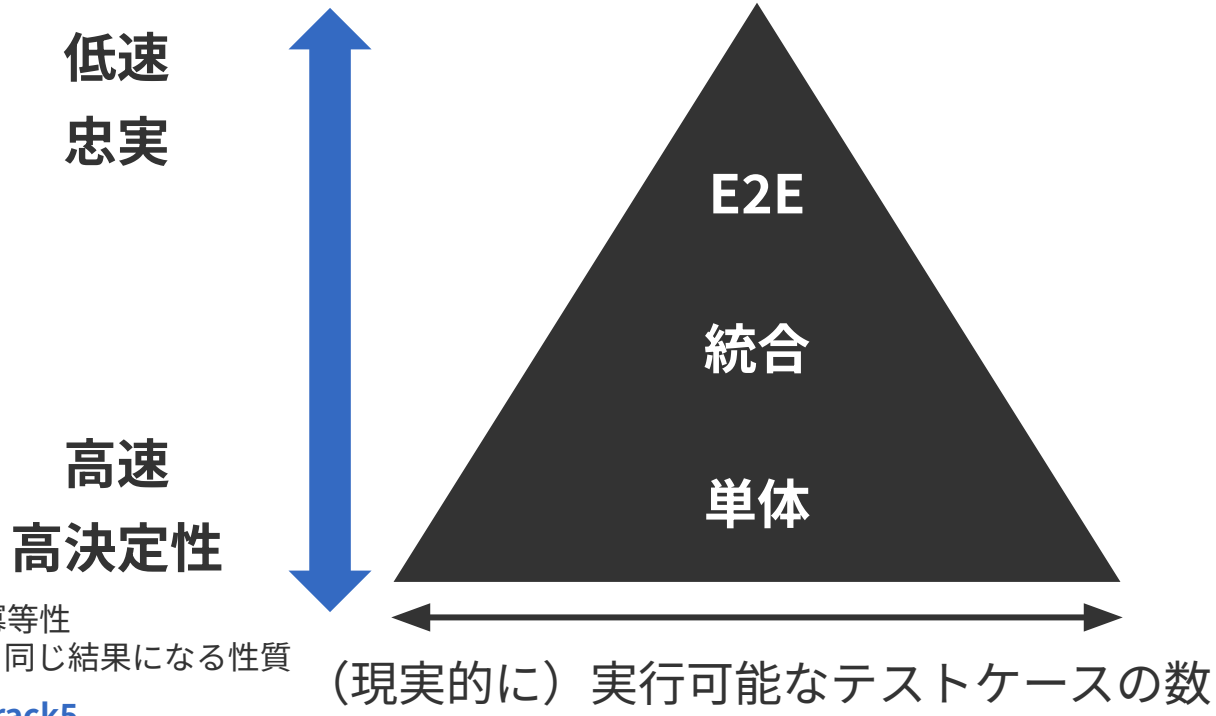
開発チームの課題として

「どこにどんなテストを書けばいいかわからない」

という声があがっていた

テストには「レベル」がある

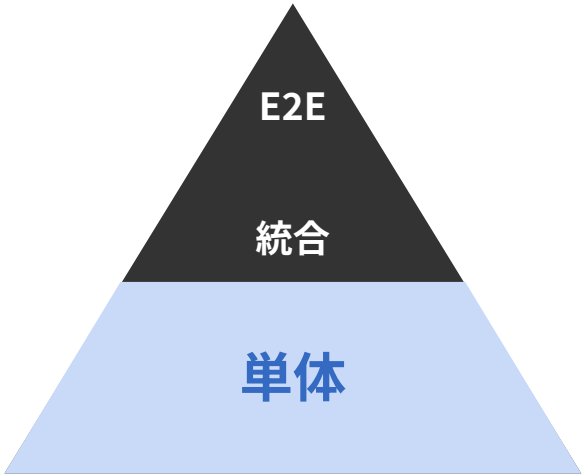
レベルごとに長所と短所がある



決定性：≒冪等性
何回やっても同じ結果になる性質

各テストレベルごとの役割

レベル低のテスト

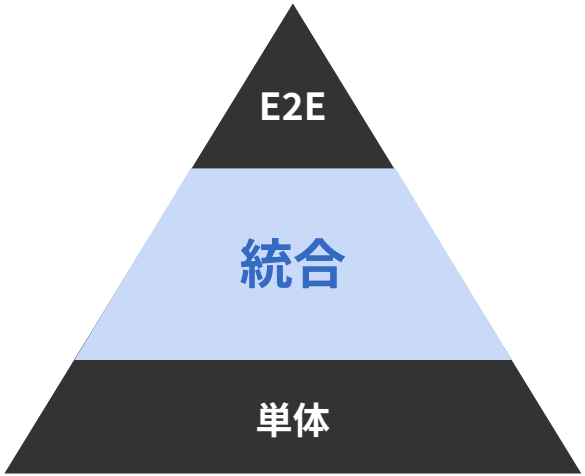


特徴	単一のプロセスで完結するため、 **実行時間が短く決定性が高い**
短所	テストで実行されるコード量が少なく、 **忠実性が小さい**
長所	**細かいテストケースをたくさん**テストできる

正常系に加え異常系をたくさんテストする

各テストレベルごとの役割

レベル高のテスト



特徴	DB等のリソースへアクセスを行う。 **実行時間が長くコストがかかるが、忠実**
短所	実行できる**テストケース数が限られてくる**
長所	テストで実行されるコード量が多く、 **忠実性が高い**

実行するテストケースを絞り

モジュール・リソース間の疎通に重点を置いたテストにする

用語の問題点

「単体テスト」「統合テスト」…

学派/考え方や個人によって定義、捉え方が異なる

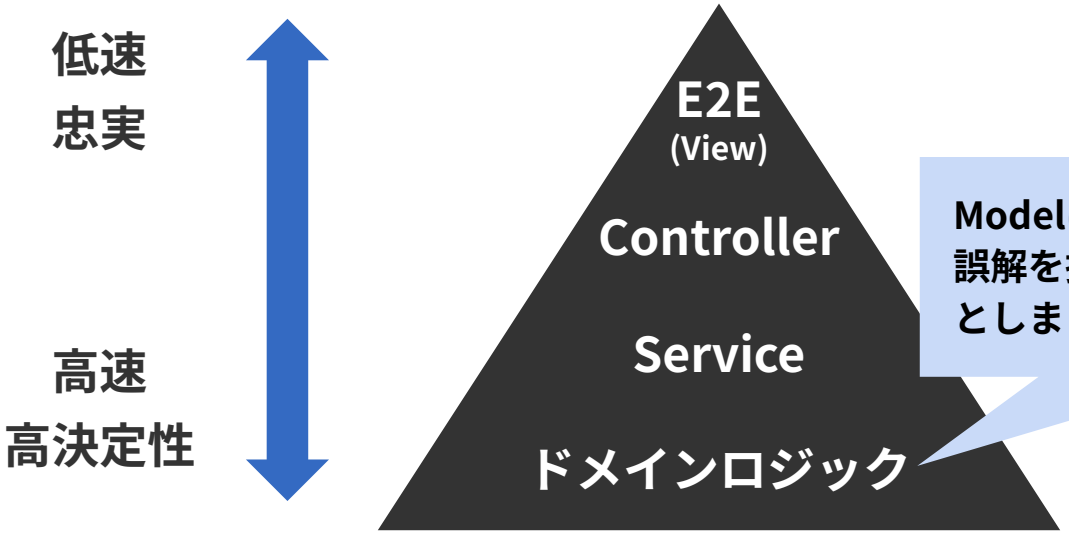
▼

実際のコードに紐つけて
考えたほうがわかりやすい

参考：https://gihyo.jp/dev/serial/01/savanna-letter/0003

コードに紐付けて考える

オープンロジの技術スタックである
Laravel（MVC+Service）にざっくり当てはめる



Modelはここに含まれると思いますが
誤解を招かないようドメインロジック
としました

各レベルごとのテストの書き方

ドメインロジックのテスト（レベル低のテスト）

DB等の外部リソースへのI/Oがない・少ないもの
正常系と異常系をたくさんテストする

Serviceのテスト（レベル中のテスト）

DB等へのリソースへアクセスを行うもの
正常系と、そのテストでしか検証できない異常系をテストする

Controllerのテスト（レベル高のテスト）

E2Eに近い領域で、実行できるテストケースが限られる
意図したレスポンスが返ることをテストする
正常系と、Controllerのテストでしか検証できない異常系をテストする

この節のまとめ

テストにはレベルがあり、それぞれ長所・短所がある

長所・短所を理解し、レベルごとに適したテストを書く

使用するリソースや実行コストで分類する、googleが提唱している「テストサイズ」という分類方法があります。今回紹介したのはこの「テストサイズ」の考え方ともいえます。

Outline

ガイドライン作成を決意するまで

書籍から学んだテストの基本

- なぜテストするのか
- 振る舞いをテストするということ
- 何をどのくらいテストすればいいのか
- **認知負荷を下げるための工夫**

ガイドライン作成のポイント

導入後の課題

まとめ・今後の課題

認知負荷を下げるための工夫

ちょっとしたことで可読性は向上する

関数の命名は検証したいことを具体的に表現する

```
// NG: 検証したい内容が不明瞭
public function test_validation_error()

// OK: テストケースをわかりやすく書く
public function test_お届け予定日を過去の日付に設定するとエラー()
```

適切なアサーションを使う

```
// NG: 認知負荷の高い例
$this->assertSame(in_array($result, 'value', true), true);

// OK: 適切なAssertionを使うと読みやすくなる
$this->assertContains($result, 'value');
```

テストが落ちたときの
エラーの文言もわかりやすくなります

Outline

ガイドライン作成を決意するまで

書籍から学んだテストの基本

ガイドライン作成のポイント

導入後の課題

まとめ・今後の課題

意気込んで作った！

初稿のレビューで撃沈

本に書いてあることをそのまま書いてしまっていた

スタッフエンジニアのレビューで

「**もっと具体的な行動に結びつくようにすべき**」との指摘が

Saved for later

[redacted] Apr 8th at 6:41 PM

概念としては、書かれている内容に特に違和感は無いです。

ガイドラインを、たとえば自分が書こうと思った場合、エンジニア向けの具体的な行動指針を中心に書くと思います。
エンジニアが、読んでいて、何を守る必要があるのか、テストを書く時に何をしなければいけないのか、というのが明確になっていると良いと思います

レビュー時の実際のslackです

「具体的な行動に結びつく」とは？

これはオープンロジではこう書く、

という具体的な書き方

「教科書の丸コピ」ではNG

厳しくしすぎない

ルールが厳しい（行動を阻害される）と、守る意欲が低下する

Why rules don’t work for some of the population

16 September 2024

Excessive regulatory burden causes economic harm and can undermine trust in government. Policymakers wishing to ease this should be more mindful of people’s differing responses to rules, says PhD candidate Ritsart Plantenga in his dissertation.

> Governments have been trying to ease regulatory burden and bureaucracy for decades, particularly from an economic perspective. But people have different attitudes towards rules, some being driven by past experiences and emotions

ライデン大学（オランダ）Ritsart Plantengaの研究

何かのルールで人々の行動が阻害されたとき、
その経験が人々にネガティブな感情を残し、
規則を順守する意欲を失わせる可能性があるといいます。
（Gigazine記事より引用）

https://gigazine.net/news/20240929-rule-experiences-and-emotions/
https://www.universiteitleiden.nl/en/news/2024/09/why-rules-dont-work-for-some-of-the-population

「教科書通り」にはできない

一方で、具体的な行動に結びつく
ある程度まとまったルールを作りたい

どうする…？

参考にしたもの

Redux Style Guide

Style Guide > Style Guide: Best Practices

Redux Style Guide

Introduction

This is the official style guide for writing Redux code. **It lists our recommended patterns, best practices, and suggested approaches for writing Redux applications.**

Both the Redux core library and most of the Redux documentation are unopinionated. There are many ways to use Redux, and much of the time there is no single "right" way to do things.

However, time and experience have shown that for some topics, certain approaches work better than

Reactでよく使われる、代表的な状態管理ライブラリの公式スタイルガイド

https://redux.js.org/style-guide/

Redux Style Guide

重要度別にPriorityがついている

Priority A: Essential

These rules help prevent errors, so learn and abide by them at all costs. Exceptions may exist, but should be very rare and only be made by those with expert knowledge of both

Priority B: Strongly Recommended

These rules have been found to improve readability and/or developer experien
will still run if you violate them, but violations should be rare and well-justified.
whenever it is reasonably possible.

Priority C: Recommended

Where multiple, equally good options exist, an arbitrary choice can be made to
rules, **we describe each acceptable option and suggest a default choice**. T
to make a different choice in your own codebase, as long as you're consistent and have a good reason.
Please do have a good reason though!

Priority	Description
A	必須。必ず守ること
B	強い推奨。可能な限り守る
C	複数の選択肢があった場合のデフォルト

この基になったというVue Style GuideはPriority Dまでありました

「テストコードのスタイルガイド」にしました

Redux Style Guideにインスパイアされ
「ガイドライン」ではなく**「スタイルガイド」**という名前に

テストコードのスタイルガイド

本書の目的

- テストコードに対する信頼度を上げ、開発品質・開発速度を向上させる
- テストコードの認知負荷を下げ、開発速度を向上させる
- テストコードに関する共通認識をつくり、開発者によってムラのないテストを作成する

重要度を定め、必ず守りたいルールとそうでないものに分ける

具体的な書き方が明確にわかるようにする

スタイルガイドの中身（一例）

絶対に守ってほしいことは「Priority: A」

モックを多用しない
各レイヤーで何をテストすればいいか
関数の命名はテストで検証したいことを具体的に書く

守ってほしいけど、事情によっては守りにくいそうなものは「Priority: B」

DBを使うテストにおいて、DBのtransactionでデータをリセットしない

現状A、Bほど重要でないが、書いておきたいものは「Priority: C」

適切なAssertionを利用する

可能な限りサンプルをつけた

すべてではないが、サンプルがすぐに用意できるものは記載し、
実際にどのようなコードを書けばよいかがわかるように

【C】適切なAssertionを用いる（assertSameを多用しない）

assertSameを使えば何でもできてしまうが、認知負荷を下げるためには適切なAssertionを用いると良い。

```
// 認知負荷の高い例
$this->assertSame(in_array($result, 'val', true), true);

// 適切なAssertionを使うと読みやすくなる
$this->assertContains($result, 'val');
```

なお、assertEqualsは厳密に型をチェックしないため使わないこと。

Outline

ガイドライン作成を決意するまで

テストの基本

ガイドライン作成のポイント

導入後の課題

まとめ・今後の課題

「テストコードのスタイルガイド」策定後、
子供が生まれまもなく育休へ…

スタイルガイド導入後半年経ち、rikutoの育休復帰を機にふりかえりを実施

実際に得られた声

ガイドラインあるおかげで、ガイドライン見てねでレビュー指摘できるようになったので、助かってます

PRを出す時、テストの書き方これで良さそうかのチェックにちょこちょこ利用させてもらっています！

毎回ではないものの、レビュー時の指摘の参考にしています。

※事前に行った、開発メンバーに対するアンケートの回答をそのまま記載しています

その一方で…

実際に得られた声②

あまり**浸透してなさそう**なとこに課題感を感じる
個人的にも**あまり意識できてない**時が多く、今見返してみるとAのライン守れてないPRレビューしたこと結構あるなという感じです…

新しく入った人として…存在は知っていたけれども
ちゃんと意識できていたかで言うとNOでした。

直近テストコード書く機会があり参照したところとても参考になったので
浸透していけるといいなと思いました

※事前に行った、開発メンバーに対するアンケートの回答をそのまま記載しています

定着していない

ルールが守りづらい以前に、ルールそのものの存在が
忘れられてしまう

スタイルガイド導入で得られた学び

「作って終わり」にしてはいけない！

作ったものの定着を
考えていくことも重要

Outline

ガイドライン作成を決意するまで

テストの基本

ガイドライン作成のポイント

導入後の課題

まとめ・今後の課題

まとめ

テストの目的は、**バグを早い段階で捕捉し、変更容易性を保つ**ということ

テストコードのスタイルガイドを作成し、開発チームに展開した

コーディングガイドラインは**具体的な行動につながる内容**にすることが大事

作って終わりにするのではなく、**定着させるための施策も重要**

今後の課題

定着させるための施策の実行

浸透させるための啓蒙活動

PR-Agent等による自動レビュー

ふりかえり・スタイルガイドの見直し

定期的にふりかえり、より開発チームに合うように
チューニングしていく

参考文献一覧

書籍

『単体テストの考え方/使い方』Vladimir Khorikov(著), 須田智之(訳), マイナビ出版
『ソフトウェアテストの教科書 第2版』布施昌弘ほか, SBクリエイティブ
『Good Code, Bad Code』Tom Long(著), 秋勇紀ほか(訳), 山本大祐(監訳), 秀和システム
『テスト自動化実践ガイド』末村 拓也, 翔泳社
『Googleのソフトウェアエンジニアリング』Titus Wintersほか(編), 竹辺靖昭(監訳), 久富木隆一(訳), オライリー・ジャパン

Web

サバンナ便り ～ソフトウェア開発の荒野を生き抜く～
https://gihyo.jp/dev/serial/01/savanna-letter/0003 ほか, 最終閲覧 2024/12/20

ありがとうございました！