ユニットテストを小さく書こう

taisei miyaji

流れ

- ユニットテストについて説明
- ユニットテストを行う際の方針
- メリット、デメリット

ユニットテストとは

- 開発者自身が作成する
- ソースコードの個々のユニットを対象とする(クラス、もしくはメソッド)
- 自動化されている

ユニットテストの目的

- コードの欠陥を早期に発見する
- 正しく変更が行われていることを保証する
- 統合テストの簡素化
- テストコードは生きた仕様書

ユニットテストの種類

- ユニットテスト(単体テスト)
  関数もしくはクラスを対象とした小さなテスト
- インテグレーションテスト(統合テスト)
  単体テストで検証した関数もしくはクラスを組み合わせて行うテスト

本題✍️

ユニットテストは小さく書こう

今回の方針であり結論
テスト対象のユニット以外を全てモック、スタブ化してテストを行う

テストの影響範囲をできるだけ小さくする

単体テストは自分のクラスの境界を越えるべきではない

- 境界を越えるテストは統合テストになってしまい、テストが失敗したときにそのコンポーネントが失敗の原因かどうか特定できなくなってしまう
- 境界を越えてDBにアクセスする場合、テストの実行時間が長くなる

テストコードは生きた仕様書である

悪い例

```
public function testUseCase(): void
{
    $entity = new Entity('testData');
    $id = 'testId'
    $repository = $this->app->make(RepositoryInterface::class);
    // あらかじめDBに保存しておく必要がある
    $repository->save($entity);

    //テスト対象関数の呼び出し
    $useCase->process();

    // ここで何らかのassertをする

    $repository->delete($id); // テストに使用したデータを削除する
}
```

ここでseederや実際のRepositoryを使用するとユニットテストでデータの保存先を意識したテストを書くことになる

理想はそのユニットテストに必要な情報全てがテストドライバ内に記載されていること
->テストドライバを見るだけで入力、期待している出力がわかる

具体的にLaravelの場合どう書くか(Mockery)

```
public function testUseCase(): void
{
    // Repositoryから返してほしいEntityを準備します。
    $entity = new Entity(
        'testData'
    );

    // RepositoryのMockを生成します。
    $repositoryMock = Mockery::mock(Repository::class, RepositoryInterface::class);

    // Repositoryの振る舞いを設定します。
    $repositoryMock->shouldReceive('findById') // findById関数が呼ばれた時の振る舞いを設定します。
        ->once() // 呼び出し回数の設定
        ->andReturn($entity); // 返り値を指定できます。ここでは先ほど用意したEntityクラスのインスタンスを返します。

    // Repositoryに$repositoryMockをDIします。
    $this->app->instance(RepositoryInterface::class, $this->repositoryMock);

    // RepositoryのDI後にUseCaseもDIします。
    $useCase = $this->app->make(UseCaseInterface::class);

    //テスト対象関数の呼び出し
    $useCase->process();

    // この後何かしら処理の結果をassertします。
}
```

メリット

- テスト実行時間の改善
- テスト失敗箇所の特定
- テストコードを仕様書にできる
- テストが担保したい内容が明確になる

Tips

テストコードをDIの仕組みを利用して書く
Laravelでいうサービスプロバイダの仕組みをテストコードに反映するとこんな感じで書けます。

```
// Repositoryに$repositoryMockをDIします。
$this->app->instance(RepositoryInterface::class, $this->repositoryMock);

// RepositoryのDI後にUseCaseもDIします。
$useCase = $this->app->make(UseCaseInterface::class);
```

テスト駆動設計との相性

テスト駆動開発とは、
開発において動作目標を先に決めておくことで開発者の不安要素を取り除くことのできる開発手法

テストを書く
↓
テストが通るようにコードを書く(汚くてもテストさえ通れば良い)
↓
テストが通る状態を維持しながらコードを綺麗にする

テスト駆動開発の場合の関心事は開発対象のクラスのみにしたい
開発対象以外をモック化することで実現できる

まとめ📝

- テストコードを仕様書にしよう
- テスト駆動開発、クリーンアーキテクチャといった手法、思想の恩恵を最大限受けよう
- テストも実際のコードと同じく疎結合かつボトムアップで書こう

おまけ

https://taisei-miyaji.hatenadiary.com/entry/2022/07/15/205958