

エンタープライズ開発でユニットテスト を普及させるために工夫した話

単体テストはエンジニアを救うLT大会

April 27th, 2022

自己紹介

- ❖ twitter: @tyonekubo
- ❖ SIer勤務
- ❖ 業務パッケージソフトの製品開発リーダー
- ❖ テスト駆動開発(TDD)を好む



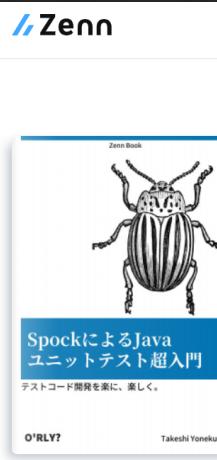
yonekubo@テスト駆動開発
@tyonekubo

Software Architect
Java/Groovy(Spock)/Kotlin/Scala
TypeScript/React
TDD/BDD/OOD/Clean Architecture
IT系の方フォロバしてます
TDDの壁を取つ払いたい/TDDに関する質問等お気軽にどうぞ

Spockで学ぶ テスト駆動開発のコツ

On May 23rd, 2021
At JJUG CCC Spring
By Takeshi Yonekubo

<https://www.youtube.com/watch?v=KZHYh2X6ZXw>



Zenn

**SpockによるJava
ユニットテスト超入門**

Takeshi Yonekubo

無料で読める本

Groovy製のテスティング・フレームワークであるSpockを使ってJavaプログラムのユニットテストを書くための入門書です。
短時間でSpockを理解しテストコードを書き始められるように、エッセンスを絞ってまとめました。

<https://zenn.dev/yonekubo/books/6f4bde620a7bac>

背景

プロダクト開発を始めるにあたり、以下の課題感があった

- ❖ ユニットテストが普及しきっていない
- ❖ ユニットテストへの取組み方が人によりまちまち

ユニットテスト普及の阻害要因

- ❖ テストコードを書くのが大変(手間がかかる)
- ❖ テストしにくい設計

やったこと

- ❖ Spockの導入
- ❖ クリーンアーキテクチャの導入
- ❖ コーディングガイドの整備

Spockの導入

Groovyなので
簡潔な記述が可能
(def、リストリテラル等)

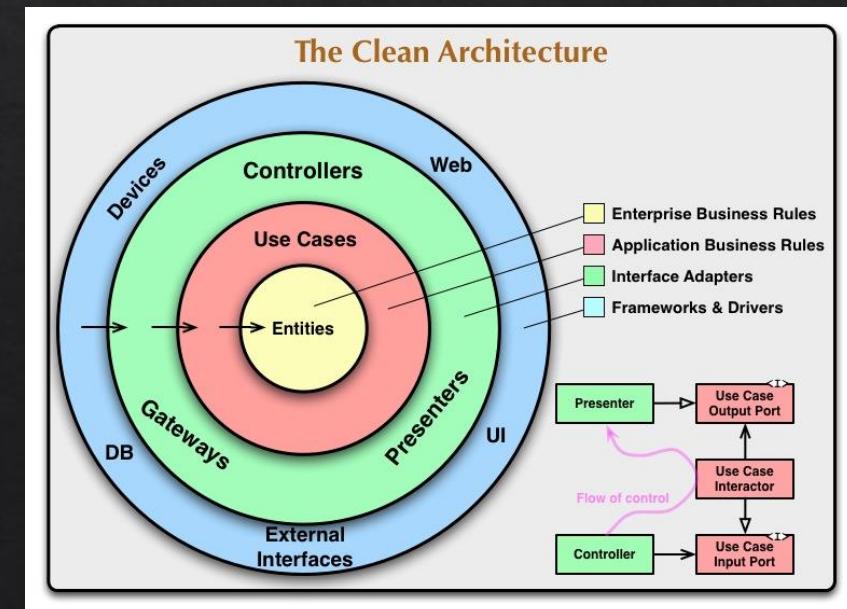
パラメータ化テストも見
やすい形で書ける

```
class SumSpec extends Specification {  
  
    @Unroll  
    def "#SUMが計算できる #formulaString should be #expected"() {  
        given: "数式をパース"  
        def formula = Formula.parse(formulaString)  
  
        and: "変数を宣言"  
        def variables = new Variables()  
        [1L, 2L, 3L].forEach { Long it -> variables.add( name: "x", BigDecimal.valueOf(it)) }  
        variables.add( name: "y", BigDecimal.valueOf( val: 4L))  
        variables.add( name: "z1", BigDecimal.valueOf( val: 5L))  
  
        when: "数式を評価"  
        def value = formula.evaluate(variables)  
  
        then: "値が正しい"  
        value.isPresent()  
        value.get() == BigDecimal.valueOf(expected)  
  
        where:  
        formulaString      || expected  
        "SUM(x)"          || 6  
        "SUM(y)"          || 4  
        "SUM(x, y)"       || 10  
        "SUM(1, 2)"        || 3  
        "SUM(-1, -2)"     || -3  
        "SUM(x, -1, y, 2)" || 11  
        "SUM(1, z1)"       || 6  
    }  
}
```

given-when-thenの
形式で整理して書ける

クリーンアーキテクチャの採用

- ◆ドメイン層から実装詳細への依存を排除し、ビジネスロジックを際立たせること、テストしやすくすることを狙ってCAを採用した



図の引用元:

<https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>

コーディングガイドの整備

- ❖ コーディングガイドに、テスト容易性に関するルールを追加
 - ❖ DIにはフィールド・インジェクションではなくコンストラクタ・インジェクションを使用する
 - ❖ 静的メソッドを乱用しない
 - ❖ など

効果(開発者の声)

テストを書くのが楽になった！

他のプロジェクトでもSpockを使いたい！



一方で、課題も

- ❖やっぱり人によってまちまち
- ❖何に対してテストを書くのか
- ❖どれだけテストを書くのか
- ❖テストファースト？ テスト駆動開発？
- ❖結果として、一部レガシー化(テストのないコード)

やったこと(2)

- ❖ テストガイドラインの作成
- ❖ カバレッジ目標
- ❖ ボイスカウト・ルール
- ❖ PR(プルリク)のテンプレート改善

テストガイドラインの作成

- ❖ ユニットテストの定義

- ❖ 層(レイヤ)を跨がない、振舞いの集合体をテストする

- ❖ コンポーネント別のユニットテスト方針

- ❖ CAの各層のコンポーネント毎にユニットテストが必須かどうかを定義

- ❖ 責務を分割し、小さなクラス(群)をテストすることを推奨

カバレッジ

- ❖ カバレッジという手段が目的化するのは避けたい
- ❖ Googleのガイドライン^(※)を参考に目安を設定
 - ❖ Exemplary(模範的): 90%
 - ❖ Commendable(立派): 75%
 - ❖ Acceptable(許容範囲): 60%

(※)Google Testing Blog:
<https://testing.googleblog.com/2020/08/code-coverage-best-practices.html>

ボイスカウト・ルール

- ❖ レガシーコードへの対応指針
- ❖ 来たときよりも美しく
- ❖ レガシーコードに対して包括的なテストスイートを作成するの
は大変
- ❖ 改修内容(仕様追加/変更)に関するテストケースは必ず書く
- ❖ 余裕があれば、関連する振る舞いに対してテストケースを書く

PR(プルリク)のテンプレート改善

- ❖ PRのテンプレートにユニットテストに関するチェック欄を設けた
- ❖ 開発者やレビューへの意識付けに効果あり

ユニットテスト

- ガイドラインに沿ってユニットテストを書いたか? [Yes/No]
- カバレッジ測定・確認を行ったか? [Yes/No]
- テストコードが少ない理由など特記事項があれば書く

効果

- ◆基準があるので、コードを書く人も、レビューアもやりやすくなった(共通のものさし)
- ◆ユニットテストに対する意識の高まり
- ◆品質向上の手段としてユニットテストを活用するチーム文化の定着

今後の課題

- ❖ テストコード自身のメンテナンス性
 - ❖ テストフィックスチャの準備コードが冗長になりがち
→ テストパターン(※)を用いたリファクタリング
- ❖ 統合テスト、E2Eテストの浸透

(※)xUnit Test Patterns:
<http://xunitpatterns.com/>



- ❖ 苦を楽にする仕組みを考えよう
- ❖ ちょっとずつ改善していこう
- ❖ ユニットテストを書くことを当たり前にしよう

ご清聴ありがとうございました

Fin.