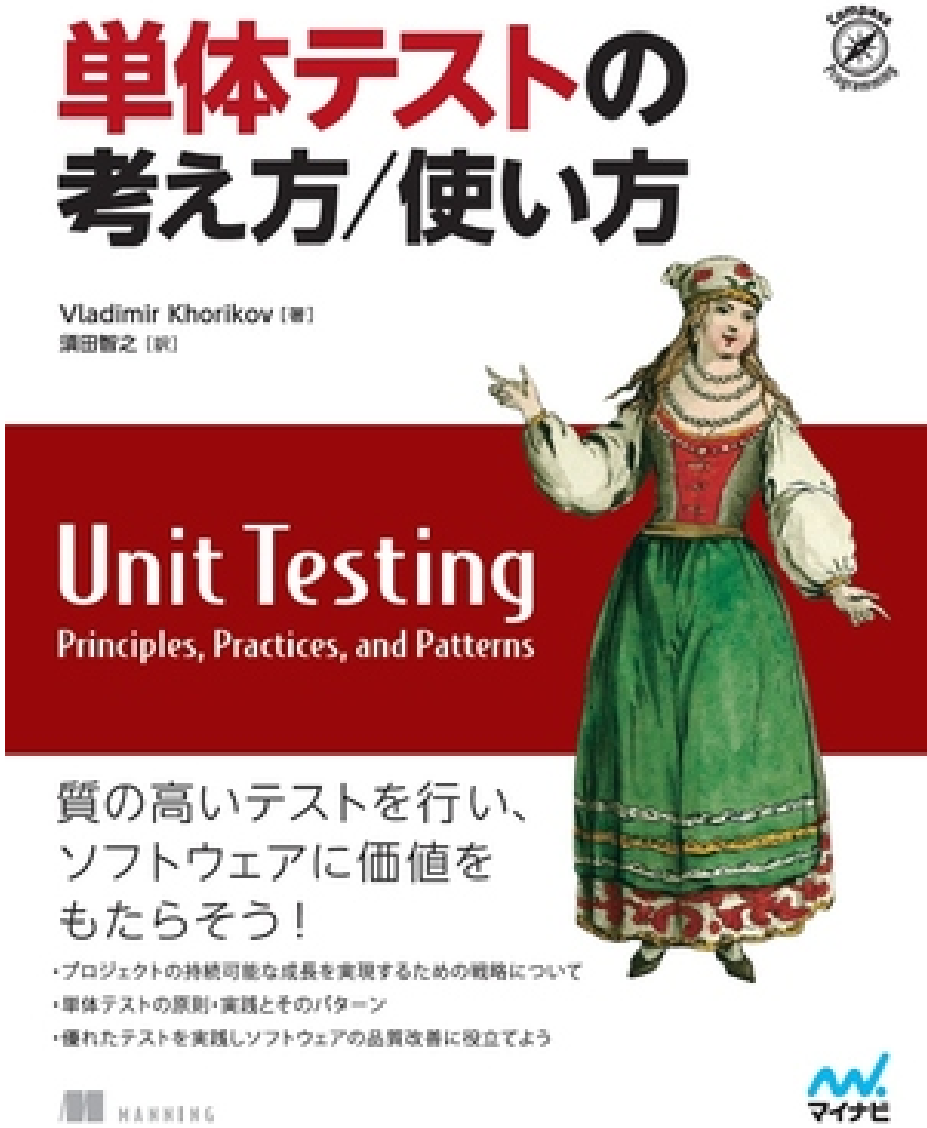
単体テストの力 : 持続可能なソフトウェア開発を目指して

伊藤 遼太

このスライドでは、主に「単体テストの考え方・使い方」にて得られる知見を参考にしています。

単体テストの目的から、具体的なプラティクスまで、効果的な単体テストアプローチが生み出す価値が説明できることを目標としています。



目次

1. 単体テストの目的

それは、ソフトウェア開発プロジェクトの成長を持続可能にすること。

単体テストを効果的に行うことの主なアウトカムとして、**設計の改善** がある。

- 単体テストを作成しづらいと感じるのであれば、そのテスト対象となるコードはなんらかの改善を必要としている可能性が高い
- 質の悪さの多くは、異なるコードが密結合していることが原因
- 単体テストが書きやすい = 疎結合になっている ≒ 良い設計

2.テストを書く際のメンタルモデル

コードは資産ではなく負債

- テストコードもプロダクションコードも負債として考えるべき
  - プロダクションコードとテストコードを分けて考える人が多いが、**テストコードもプロダクションコードと同じく保守していくもの。**
  - プロダクションコードを書く際に気をつけていることをテストコードにも適用していくべき
    - ex) 良い命名を考える、共通のロジックは抽象化して切り出す、仕様変更による修正が最小限になるようにする、など

コードは資産ではなく負債

- テストコードも最小限にすることが重要
  - テストケースは多いほど良いと思っている人もいるが、この考えは誤り。コードは資産ではなく負債だから。
    - コードが増えれば、ソフトウェアにバグが持ち込まれる経路が増えることとなり、プロジェクトを維持するコストもさらに高くなってしまう

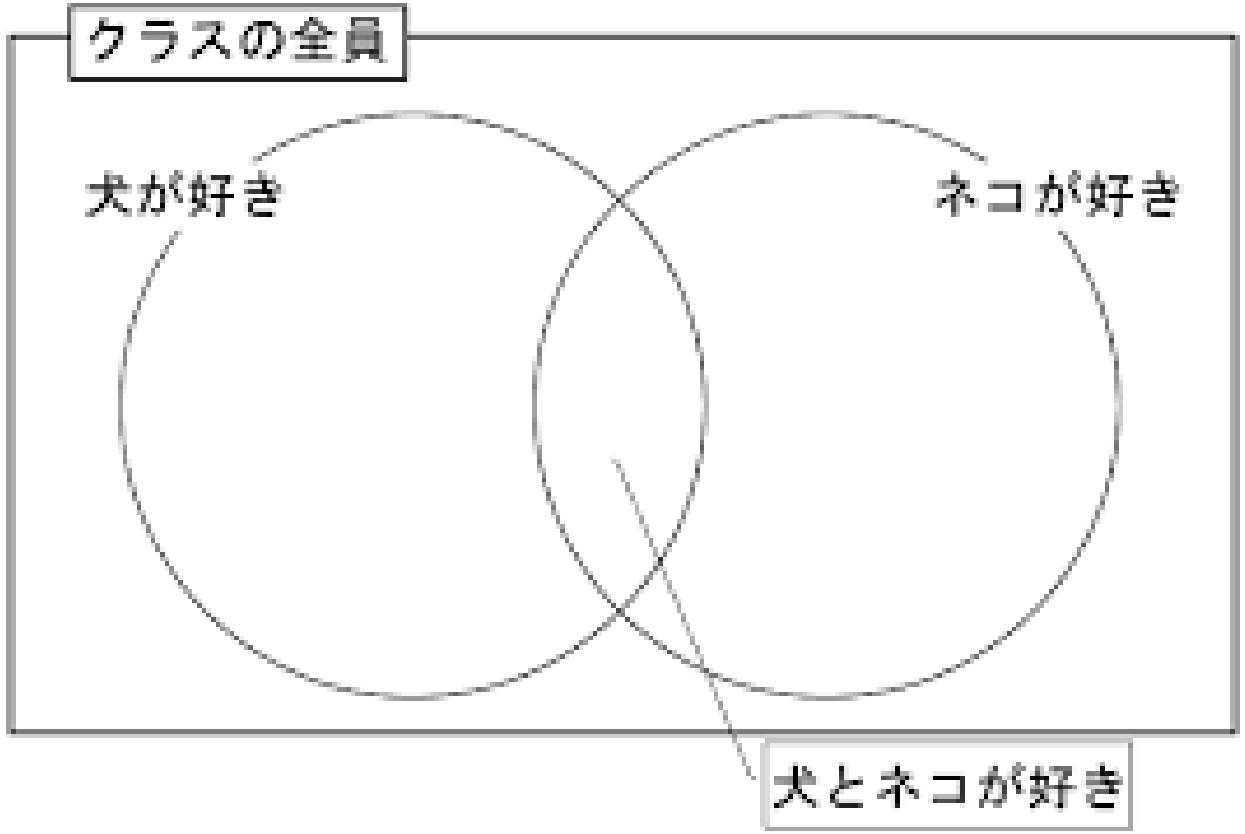
優れたテストスイートの特徴

1. 開発サイクルに組み込まれている
2. コードベースの重要な部分だけが対象（主にビジネスロジックやドメインモデル）
3. 最小限の保守コストで最大限の価値を生み出すように設計されている

3. テストの手法とトレードオフ

良い単体テストを構成する４本の柱

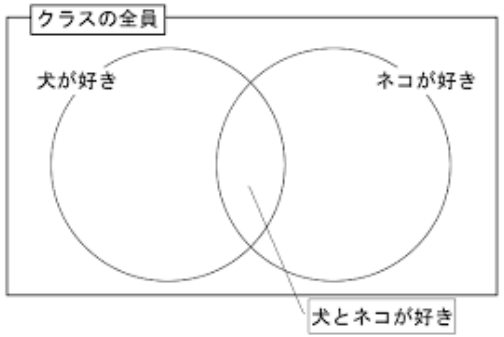
1. 退行（Regression）に対する保護
2. リファクタリングへの耐性
3. 迅速なフィードバック
4. 保守のしやすさ



退行（Regression）に対する保護

プロダクションコードが**間違っているのにテストが落ちてくれない**状況に対する保護

> ex）ライブラリのバージョンアップを行ったが、そのライブラリを
> テスト時にモックしていたためにテストが落ちなかった
> → プロダクションでバグが起きた

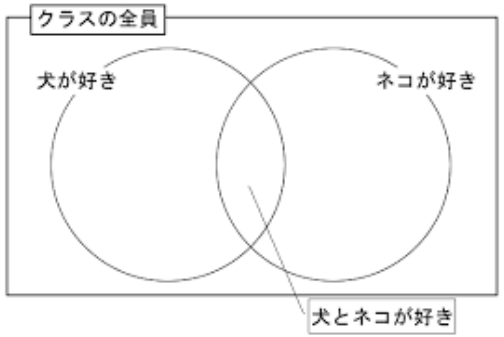


効果的なアプローチ

- **なるべく多くのプロダクションコード**を実行するようにする
- 最も適したテスト : **End-to-End テスト**

リファクタリングへの耐性

リファクタリングを行った際に**偽陽性**（プロダクションコードの振る舞いは正しいのにテストが落ちてしまうこと）を生み出してしまうこと



ex)
- Aの関数のテストの際に、Aの中で呼ばれているBをモックしており、
  Bのインターフェースを変更したらAの関数のテストが落ちてしまった
- コンポーネントで出力されるDOMを文字列でアサーションしており、
  spanタグをpタグにしたらテストが落ちてしまった（振る舞いは問題ないのに）

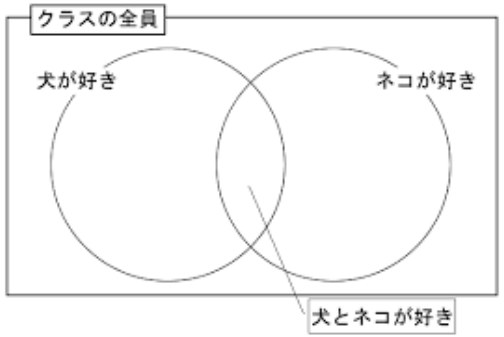
リファクタリングへの耐性

効果的なアプローチ

- **「観測可能な」** 範囲のみを検証すること。テスト対象が公開している物のみを使うこと。
- **「実装の詳細」** をテストしないこと。暗黙に呼ばれているクラスや、出力される SQL や HTML の文字列等の、実際のコードの**振る舞い**以外をテストしないこと

迅速なフィードバック

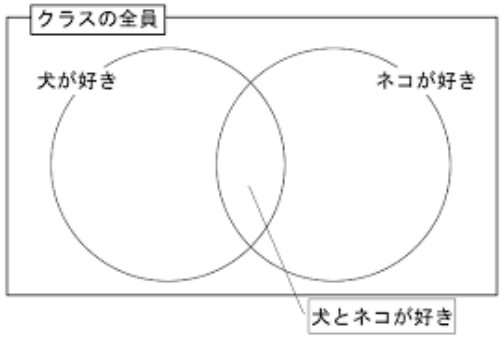
- 迅速なフィードバックを測る指標は、テストの実行時間。
- E2Eテストは実行時間が長く、外部依存等がない単体テストは実行時間が短い。



保守のしやすさ

保守のしやすさは以下の二点で評価できる

1. テストケースを**理解すること**がどれくらい難しいのか
    - テストケースのサイズによって変わる。一般的に、コードの量が少ないほど読みやすくなり、必要に応じて変更することも簡単に行える
2. **テストを行うこと**がどのくらい難しいのか
    - いろんなものを起動させないと実行できないような場合など



理想的なテストの探求

- ４つ全てをバランス良く満たすテストを書くことは**不可能**である。
  - 「保守のしやすさ」を除く３つの柱は互いに排反する性質だから
- ３つの中でも、**リファクタリングへの耐性を落とすことは現実ではできない。**また、保守のしやすさも落とすことはできない。
- よって、**「退行（Regression）に対する保護」**と**「迅速なフィードバック」**のトレードオフを適切に理解し、それぞれを**バランス良く満たす**テスト構成にするべきである
- その結果、**テストピラミッド**のプラクティスが適用されることが多い

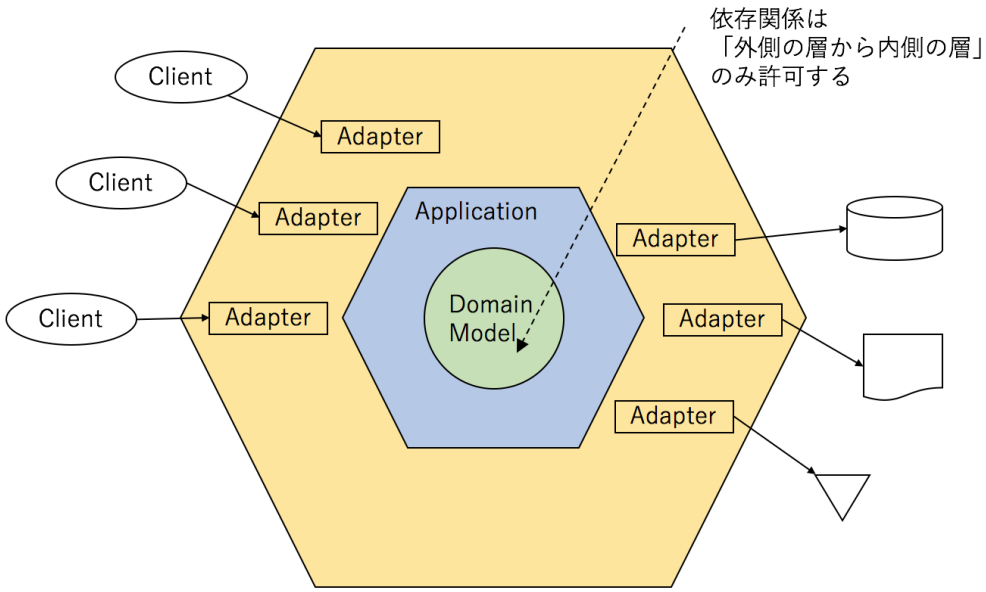
4. 設計と単体テスト

良い設計が良い単体テストを導く

ヘキサゴナルアーキテクチャ

- 内側: ドメイン層
- 外側: アプリケーション層

ドメイン層にビジネスロジックをよせ、単体テストを書きやすくすることで自然に責務の分離ができ、疎結合で良い設計に近づく
外部システムとのコミュニケーションはモックに置き換える



この本におけるアーキテクチャ

この本ではヘキサゴナルアーキテクチャを採用しており、3層のレイヤーを定義している。
多すぎるレイヤー化はアンチパターンとしている。レイヤーを細かく分けると、それぞれのレイヤーに対するテストを書きたくなってしまうため。

5. モックの利用と壊れやすさとの関係

偽陽性を生むテスト

大抵、モックは**実装の詳細**に関心を持つことになり、実装の詳細をテストすると壊れやすいテストになる

振る舞いを変えずに内部実装を変える「リファクタリング」で偽陽性が発生する

モックの考え方 : ロンドン学派と古典学派

ロンドン学派： 一つのクラスをテスト対象とする。他は全てモックする
古典学派： 「振る舞い」をテスト対象とする。テストケース同士で共通の依存がなければモックは必要ない

- ロンドン学派のアプローチは壊れやすいテストになる可能性がある
- この本は、「古典学派」の立場。モックした方が良いテストになる場合もあるが、**基本的にはモックしない**方が望ましいという考え

TIPS : ドメイン層の隔離

- ドメイン層の中でプロセス外依存（永続層など）にアクセスしたりすると、テストではそこにモックを差し込まなければ行けなくなる
  - モックを差し込まなければいけない = 内部実装を読んで、どこでどのオブジェクトが使われているか、どういう関数が呼ばれるか、というのを知っている必要がある
  - 内部実装とテストケースが紐付く = リファクタリングへの耐性が低くなる
    - **テストコードを理解するためには内部実装を知っていなければいけない ≒ 認知負荷が上がる ≒ 保守のしやすさが低くなる**
    - **ビジネスロジックのテストの質が低くなる ≒ アプリケーションの設計の質が低くなる**

6. プラクティス

読みやすいテストのために

- AAA パターンを用いて記述する（準備 (Arrage), 実行 (Act), 確認 (Assert))
  - 空白行やコメントを使ってわかりやすくする
  - １つのテストケース内で同じフェーズが複数でてこないようにする
- そのテストケースの中でのテスト対象を特定の命名を使う
  - sut (System Under Test) など
- 全てのプロパティをアサーションするよりもオブジェクトと等価なことがテストできるようにする

読みやすいテストのために

```
fun test() {
    // arrange
    val stub = StubObject("hoge")
    val sut = SystemUnderTestEntity.new( hoge = stub )

    // act
    val sutResult = sut.execHoge()

    // assert
    val expected = SystemUnderTestEntity.restore( hoge = "hoge" )
    assert(sutResult == expected)
}
```

テストの3つの手法

	出力値ベース・テスト	状態ベース・テスト	コミュニケーション・ベース・テスト
リファクタリングへの耐性を維持するのに必要なコスト	低い	普通	普通
保守のしやすさを維持するのに必要なコスト	低い	普通	高い

出力値ベースのテストが一番コストが低い。

出力値ベース・テスト ： 入力に対する出力を検証する。純粋関数のテスト
状態ベース・テスト ： 副作用を起こす関数をよんで、その後の状態（DBなど）を検証する
コミュニケーション・ベース・テスト ： モックをさして、そのモックが呼ばれたかどうかを検証する

E2Eテスト

- 結合テスト : できるだけモックしない （状態ベース・テスト / コミュニケーション・ベース・テスト）
- １件の一番長いハッピーパスと単体テストでは検証できない全ての異常ケースを検証することが適切だと考えられている

TIPS

- １つで網羅できない場合は、いくつかに分けて検証されるようにする
- 意味のないテストは作成しない
  - 不正なメールアドレスが渡された場合、など、仮にチェックが漏れていたとしてもドメインモデルで弾かれることがわかっているようなテストケースは作成する意味がない
  - 質の悪いテストケースを作成するくらいなら、テストケースを作成しない方がまだマシ

単体テスト : Application 層 / Infrastructure 層

- テスト対象によって、適切にモックをさしての検証を行う。
- 状態ベース・テストやコミュニケーションベーステストになる
- ex) Controller / UseCase / Repository

TIPS

- 基本的にモックをさしてのテストになるため、壊れやすいテストになる。そのため、なるべくこのテストが膨らまないようにする
- ロジックがない場合は作成する必要なし

単体テスト : Domain 層

- 出力値ベーステストが望ましい。Immutable なクラスなどであれば状態ベーステストになる
- 原則、モックは使用しない
- ex) Entity / Domain Service

TIPS

- 他の層にビジネスロジックが漏れ出ないように、重要なロジックはこの層に閉じ込め、十分にテストが書けるようにする

完

ありがとうございました。