単体テストのすすめ

A recommendation of unit testing

KOMATSU Seiji (comutt)
atWare, Inc.
September 14, 2012

skomatsu [at] atware.co.jp
facebook.com/comutt
@comutt

About

- 名前: 小松 聖司
- 職業: システムエンジニア
- 職歴: 株式会社アットウェアにて、
  Web アプリ開発に携わること4年目
- 言語: 日本語, 英語, Java, Python, etc

Note

- ソースコードは、（ハンズオン完成版ですが）以下で公開しています。
- https://github.com/comutt/unittest-study
- まったくもって Timetable 通りには進みませんでした。（タイムアップ）

Purpose

単体テストを
おすすめ・布教する

Timetable

- 19:00 - 19:30
  単体テストについて説明
- 19:30 - 20:00
  単体テストを実際に書いてみよう
- 20:00 - 20:15
  自動テスト（CI）について説明
- 20:15 - 20:30
  自動テスト（CI）の実演
- 20:30 - 20:40
  まとめ
- 20:40 - 21:00
  質疑応答、撤収

Target

- 対象
  - Lv.0 単体テストを知らない方
  - Lv.1 単体テストの書き方を知らない方
  - Lv.2 単体テストの書き方は知ってるけど、書くメリットがわからない方
- 非対象
  - Lv.3 単体テストを書くメリットは理解してるけど、面倒な方
  - Lv.4 ばりばり単体テストを書いてる方

What is Unit Test?

Unit Test?

- ユニット（モジュール、クラス、メソッド）単位でテストするからユニットテストといいます
  （日本語訳: 単体テスト）
- アプリケーションを横断的・全体的にテストする結合テストに比べて、非常に小さい、軽量なテストです

Principle

モジュールの数だけ、
テストコードを書きます

Principle

パターンの数だけ、
テストコードを書きます

Example 1

コード

```
int sum(int n, int m) {
    return n + m;
}
```

テストパターン

- 1 + 1 = 2 // 正常系1
- -1 + -1 = -2 // 正常系2

Example 2

コード

```
// n を m で割った値を返す
int divide(int n, int m) {
    // 0除算防止
    if (m == 0)
        return 2147483647;

    return n / m;
}
```

テストパターン

- 9/ 2 = 4 // 正常系1
- -5 / 3 = -1 // 正常系2
- 1 / 0 = 2147483647 // 異常系

Point

- テストコードは正常系だけではなく、異常系も書きます
  - アプリが「予期せずエラーを吐いて終了」そんな経験ありませんか？
    - テストコードを書けば、その可能性をぐんと減らせます

Point

- テストコードは、適切な粒度で書きます
  - 実は何気に難しいスキルです
  - パターンを網羅することを意識しすぎると、冗長すぎるテストコードが完成します
  - テストコードがないことに比べれば1000倍良いですが、保守性を低下させるなどの弊害が生じます

Principle

テストコードは、

実装ではなく

仕様に基づいて書く

Reason

- 実装に基づいてテストコードを書いてしまうと、仕様との相違を検出できません

Merit

- ソースコードレベルで仕様が明確になります
  - 仕様書と辻褄の合わないソースコード、経験ありませんか？

Merit

- 機能追加やリファクタリングが容易になります
  - 機能追加・リファクタリングしたら、動作が変わってしまう（デグレ）かもしれない、という不安・リスクから解放されます

Optional Merit

- 引き継ぎのコストが下がります
  - テストコードで仕様が分かること
  - ソースコードを修正するリスクが低いこと
  - が主な理由です

Principle

テストコードは
こまめに書く

Reason

- 間違っても、「アプリケーション組み上げたらまとめて書く」なんて考えてはいけません
  - テストによって見つけられるはずの早期不具合発見ができなくなります
  - 実装を忘れてしまうリスクがあります
  - 往々にして、一気にテストコードを書くのは「苦痛」です

More

- できる限り、関数・メソッド単位で
  テストコードを追加していきましょう
- つきつめると、
  TDD (Test Driven Development)、
  ないしはそれに近い形になります

Cycle

実装（修正）する
→テストコード書く
→テストを実行する
→実装（修正）する

Principle

テストコードは、
常にアップデートする

Reason

- 「仕様に基づいて書く」のだから、
  仕様がアップデートされるたびに、
  テストコードがアップデートされなければ
  なりません
  - そうしないと、テストコードが腐ります
  - 自動テスト（CI）を活用します（後述）

1 Minute Break

How to write Unit Test?

Unit Testing Library

- 単体テストのライブラリが存在するので、言語・開発環境に合わせて選択します。
  （標準で用意されている言語・開発環境もあります）
- 一例
  - C: CUnit, CCUnit
  - C++: CppUnit
  - .NET Framework: Visual Studio 標準, NUnit
  - Java: JUnit, TestNG
  - iOS: Xcode 標準
  - Android: Android SDK 標準 (JUnit)
  - PHP: PHPUnit
  - JavaScript: Jasmine, QUnit

How to Write?

- 言語、ライブラリ次第なので、
  それに併せて覚える必要があります。
- 言語習得するのと同じように、
  テスト記法を習得する必要がある、
  と考えればよいと思います。

Try with JavaScript

- 本日は JavaScript で
  単体テストを書いてみます
- JavaScript を選んだ理由
  - とりあえずブラウザがあれば動くためです
  - 「単体テストってどういうもの？」
    を理解するためには、言語は関係ないです

Jasmine

- JavaScript の単体テストに使うライブラリは「Jasmine」です
  - 以前に函館で Android ICS の講演をしてくださったわかめさん (@vvakame) におすすめされたので使ってみることにしました
  - 「Jasmineのここが優れているよ！」という趣旨ではありません

Jasmine

- Jasmine を導入するためには、
  - 公式サイトから Standalone Release 版を取得
  - ruby gems で gem install jasmine
  - github リポジトリから clone
  - のいずれかの方法でライブラリを取得します

  - 今回使用するのは、
    Standalone Release 版です

Jasmine

- Jasmine 公式サイト
  - http://pivotal.github.com/jasmine/

Downloads

- The Standalone Release is for simple, browser page, or console projects
- The Jasmine Ruby Gem is for Rails, Ruby, or Ruby-friendly development
- Other Environments are supported as well

Jasmine

```
jasmine-1.2.0/
├── SpecRunner.html
├── lib/
│   └── jasmine-1.2.0/
│       ├── MIT.LICENSE
│       ├── jasmine-html.js
│       ├── jasmine.css
│       └── jasmine.js
├── spec/
│   ├── PlayerSpec.js
│   └── SpecHelper.js
└── src/
    ├── Player.js
    └── Song.js
```

テスト実行用HTML

Jasmine ライブラリ

テストコード配置先

ソースコード配置先

Run Jasmine

- お好きなブラウザで、
  SpecRunner.html を開いてください。

- 上記のような画面が出力されればOKです。

Jasmine Spec

- Jasmine のテストコードを記述したソースファイルは、決まりではありませんが、<テスト対象js名> + Spec.js とすることにします。
- Jasmine に限らず、ほかの言語でも、このような規則で、テストコードファイルの名前を付けるのが一般的です。

Jasmine Spec

- Jasmine のテストコードは、
  おおざっぱに言って以下のように記述します。

```
describe(“テスト説明1”, function() {
  // テスト対象クラスのインスタンス
  var target;

  beforeEach(function() {
    // インスタンス生成
    target = new TestTarget();
  });

  // テスト実行
  describe(“テスト説明1.1”, function() {
    it(“テスト説明1.1.1”, function() {
      expect(target.doSome()).toEqual(“期待値”);
    });
  });
});
```

Jasmine Spec

- describe(“テスト説明”, function(){...});
  - 第一引数は、「これが何のテストか」という説明文字列です。
  - describeがネストされると、テスト結果もネストされて表示されます。
  - クラス名やメソッド名など、適切な説明文字列を指定します。

Jasmine Spec

- it(“テスト説明”, function(){...});
  - 第一引数は、describeと同じです
  - こっちは、「どのパターンのテストか」という説明文字列にするのがおすすめです。

Jasmine Spec

- expect(target.doSome()).toEqual(“期待値”);
  - 実際にテストを実行する文です。
  - 上記の場合は、doSome() が “期待値” を正しく返すかどうか、のテストです。
  - toEqual以外にも、trueかどうかを検査するtoBeTruthyなどがあります。

Try Jasmine

- 実際に、Jasmine を使ってテストコードの作成をしてみましょう。
- もちろん、ソースコードの実装もしてもらいます。

Try Jasmine

- SpecRunner.html と同じところに、GreetingSpecRunner.html があります。
- これをブラウザで開くと、テストが失敗します。
- これを成功するように、実装およびテストコードの作成を行ってください。
- なお、 Greeting.html が、テスト対象のソースコードを使ったサンプルアプリです。

Try Jasmine

- src/Greeting.js がソースコード、spec/GreetingSpec.js がテストコードです。
- src/Greeting.js は、肝心要の部分が全て未実装です。
- src/Greeting.js は、テストケースが1つ書かれています。
  このテストケースが成功したら、
  テストケースを最低2つ追加してください。

Specification

- 仕様 ver.1 は以下の通りです。
- 午前の時間は「おはよう」を表示する
- 午後の時間は「こんにちは」を表示する
- 夕方の時間は「こんばんは」を表示する
- 未定義の時間の場合は、「エラー」を表示する
- getGreeting の第1引数には、以下が渡される
  - 午前: “morning”
  - 午後: “afternoon”
  - 夕方: “evening”
  - 第1引数に未定義の時間が渡されたときは、undefined を返却する
- 仕様 ver.1 では、第2引数 name を無視する

Know How

- テストは Greeter クラスの getGreeting に対して行います。
- Java Script では、引数を省略することができます。（省略時、undefined が渡される）

Test Patterns

- 仕様 ver.1 で必要なテストパターン（例）
  - (“morning”) -> “おはよう”
  - (“afternoon”) -> “こんにちは”
  - (“evening”) -> “こんばんは”
  - (“不正な文字列”) -> undefined

Run in Browser

テストが全て通ったら、
ブラウザでも確認してみてください。

当たり前ですが、問題なく動くと思います。

Specification

- 仕様 ver.2 は以下の通りです。
- 名前が指定されたときは、「〜さん、〜」を表示する。
- getGreeting の第2引数 name には、名前が渡される。
- 名前が指定されない場合は、ver.1 と同じ挙動をする。

Run in Browser

テストが全て通ったら、
ブラウザでも確認してみてください。

Test Patterns

- 仕様 ver.2 で必要な追加テストパターン（例）
  - (“morning”, “中村”) -> “中村さん、おはよう”
  - (“afternoon”, “山中”) -> “山中さん、こんにちは”
  - (“evening”, “黒田”) -> “黒田さん、こんばんは”
  - (“不正な文字列”, “田中”) -> undefined

How do you feel?

- いかがでしたでしょうか？
- 「仕様に合わせてテストを書き、テストが通るように実装する」という一連の流れが、なんとなくお分かりいただけたでしょうか。
- テストを書くというのは、この作業の繰り返すことです。

5 Minute Break

Continuous Integration

Continuous Integration (CI)

- 繰り返しになりますが、
  テストコードを書くのは一回きりではありません。
- 実装を続ける限り、
  テストコードの作成と実行を続けなければ
  意味がありません。
- そのために、
  継続的な統合（Continuous Integration; CI）
  と呼ばれる試みが必要になります。
- この会では、継続的な統合を「自動テスト」
  として呼ぶことにします。

What is CI?

- 日本語訳の「継続的な統合」は、
いまいち意味が掴めない方が多いのではないでしょうか。
- 簡単にいうと、ソフトウェアの品質を維持するために、継続的に、高頻度で品質管理(QC)を行う仕組みです。
- といってもまだ分かりづらいので、
さらに単純に言うと、定期的にテストコードを実行する仕組みです。

CI System

- CIサーバは、リポジトリからソースコード全体を取得し、常にテストを実行する役割です。

CI Notifaction

- CIサーバは、ビルド、テスト実行後、その結果を通知することができます。

Merit

- ビルドやテストが自動ですべてなされるため、
  テストを全て実行するコストが
  基本的にかかりません。（初期設定のみ）
- 定期的にビルドされるため、
  万が一ビルドが壊れてもすぐに
  開発者（チーム）が知ることができます。
- また、どのコミットで、どの部分が壊れたか、
  が簡単に分かります。

Merit

- プロジェクトが大きくなれば大きくなるほど、テストコードも増えます。
- それらを、人の目だけでメンテナンスすることは不可能です。
- 全テストの実行は、機械的に実行するのが一番効率的です。
- なので、全テスト実行は、素直にコンピュータにお任せしましょう。

Jenkins

- CI用ソフトウェアはいくつかあります。
- その中でも、使いやすいと定評のある、Jenkins がおすすめです

Jenkins

様々なプラットフォームに
対応しており、
専門的な知識がなくても、
簡単に導入できるのが
特徴です。

Jenkins

- Jenkins のインストール・設定方法は、
  ここでは触れません。
- ほとんどのプラットフォームで動作するので、
  お試しで動かすだけなら
  どこで動かしてもよいです。
- プロジェクトで本採用するときは、
  きちんとCIサーバ用PCを構築しましょう。

Jasmine on Jenkins

- Jasmine のテストケースを、
  Jenkins で実行することは可能です。
- ここでは、
  Rhino + Jasmine + jasmine-reporters
  にて動かしてみます。
- 興味のある方は、 GitHub をご覧ください。

Jasmine on Jenkins

実演タイム

Conclusion

Conclusion

- テストコードを書くのは当たり前、
  と考えるようにしてください。
- テストコードを書くコストは当然必要です。
- が、それも、品質の良いソースコードを書くのに必要なコストです。
- テストコードを書くコストを削減することは、ソフトウェアの品質を自分から下げていることに他なりません。

Conclusion

- テストコードを書かずに、
  実装後の結合テストでテストを
  全て行う、というのはよくありません。
- テストコードを書けば、
  実装中にバグを早期に発見でき、
  リファクタリングも安全に行うことができます。

Conclusion

- アプリケーションを修正するコストは、
  実装してからの経過時間が増すほど増える傾向にあります。
- その増えるコストを無視して、
  「テストコードによって実装コストが増え、全体のコストが増えてしまう」
  と考えるのは短絡的といえます。

Conclusion

- テストコードを書くことは、
  チーム開発においてより威力を発揮します。
- が、チーム内でテストコードを書く人、
  書かない人が混在していては意味がありません。
- テストコードは、「実装者全員が必ず書かなければならない」と、あらかじめコミットすることが大切です。

Conclusion

- （まとめが5枚目になった。。。）
- テストコードを書くためのコストは、学習初期はある程度大きくなりますが、数をこなすことで、要領が分かってきます。
- 「テストコードを書かないと不安だ。書かないなんてあり得ない。」と思えれば、あなたは数段レベルアップしています。そこを目指してください。