

ブラックボックスとしての同値分割だけでは
上手く外部設計できないパターン

TDD 道場 #19

とある道場の
乱取稽古
TDD Dojo



BluewaterSoft

2014/5/24 biac



わんくま同盟 名古屋勉強会 #31

スピーカー紹介: biac as 山本 康彦

- 宇宙世紀以前の生まれ
スプートニク1号より3ヶ月ほど前
- 最初は **HONDA**
クルマの設計/研究を10年くらいやってた
- 今は **BluewaterSoft**
を名乗ってアプリ開発とか技術解説記事とか
- 「**NUnitの全貌**」 ⇒
CodeZine 2012/4

ホーム ニュース 記事 注目ブックマーク コミュニティ デブサミ
C# | Java | VB.NET | C++ | PHP | Ruby | Perl | JavaScript | SQL | Adobe | 言語一覧

C# 開発/設計/テスト

NUnitの全貌 ~ 基本から、最新バージョンの新機能まで

C#で始めるテスト駆動開発入門(3)

biac [寄] 2012/04/13 14:00

いいね! 27 +1 8 BI 85 ツイート 80

ダウンロード サンプルソース (C#) [29.84 KB]

ダウンロード CsTdd03_20130403.zip [29.74 KB]

1 2 3 4 5 6 7 8

NUnitの最新バージョン2.6の主な機能を解説します。普段からNUnitを使っている開発者でも、「こんな機能があったのか!」と驚くようなことがきっとあるでしょう。

はじめに

C#でTDDしている開発者にとって、おそらく一番なじみのあるテストングフレームワークはNUnitでしょう。今年の2月に、そのVersion 2.6が正式リリースされました。2.x系はこれが最後のリリースとなり、次はVersion 3になるとされています ([NUnit Roadmap](#)参照)。この機会に、NUnitの基本機能から、2.6で追加された新機能まで、全体を一通り把握しておきましょう。

■ NUnitの最新版バージョン2.6のテストランナー (nunit.exe)



TDD = テスト ファースト + リファクタリング

- テスト ファースト: **RED** と **GREEN** の繰り返し
- リファクタリング: **GREEN** を維持したまま実装を改善

失敗するはずのユニット テストを1つ書き、
失敗することを確認 (=RED)



ユニット テストに通るだけの実装を追加し、
成功することを確認 (=GREEN)

TDD 3原則 by Robert C Martin

• [Article S. Uncle Bob. The Three Rules Of Tdd](#) (2005)
より。

※ 実質は「テスト ファースト 3原則」

1. 失敗するユニットテストを成功させるためにしか、プロダクトコードを書いてはならない。
2. 失敗させるためにしか、ユニットテストを書いてはならない。コンパイルエラーは失敗に数える。
3. ユニットテストを1つだけ成功させる以上に、プロダクトコードを書いてはならない。



今年のテーマ

- テストファーストに必要なスキルがある。
それは…

メンツドの外部設計 (external design)

メソッドの外部設計 → テスト ファースト

外部設計 external design

- ・ 外観
メソッドのシグネチャを決定
- ・ 反応
メソッドの、引数に対する返値や (外から見える) 副作用。

OOP風にいえば「メッセージに対する (外部から見た) 振る舞い」



テスト ファースト

外部設計からひとつ取り出して、
ユニット テストに変換
⇒ それに対応する実装をしたら、次へ

外部設計 → 内部設計

- メソッドの外部設計

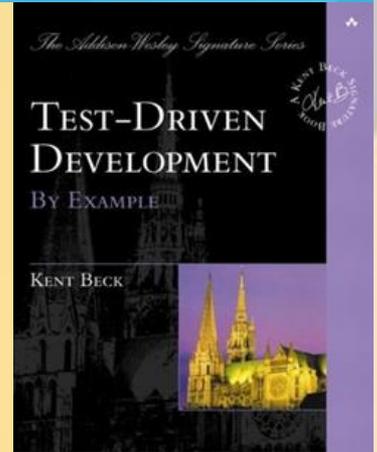


メソッドの内部設計 (=実装)

- テストファーストとは、
外部設計 ⇒ 内部設計
という**当たり前の手順**を、
効率よくやっているだけに過ぎない。

TDD における外部設計

- 外部設計の方法は規定なし
TDD の考案者 Kent Beck の本では、頭の中だけだったり、To Do リストに書き出したり



- でも、大事!!

Kent Beck レベルでは、たぶん…

「そんなん、できてるだろ、常考!」

でも日本の多くの開発者は、そんな訓練をやってない

余談: TDD したら工数がベラボーに増えちゃった orz

- 今まで当たり前の手順をしてなかった
メソッドの外部設計⇒内部設計 という手順を踏んでいなかった (外部設計をしていなかった) ので、
テストファーストのREDの分が工数増
- 今まで綺麗なコードを書いてなかった
リファクタリングで工数増
- あと大抵は、訓練不足部隊の実戦投入
↑ 戦う前から負け戦が確定してるw

前は…

- 入出力表でメソッドの反応を定義
 - 例題: 単偶数(半偶数)
 - 演習: FizzBuzz
- 入出力表だけで表現できるシンプルなもの
- ブラックボックスとしての同値分割だけ

前回の回答例 (FizzBuzz)

- 発言の順番(1,2,3,...)を入力とし、発言する文字列を出力とした

外部設計 external design

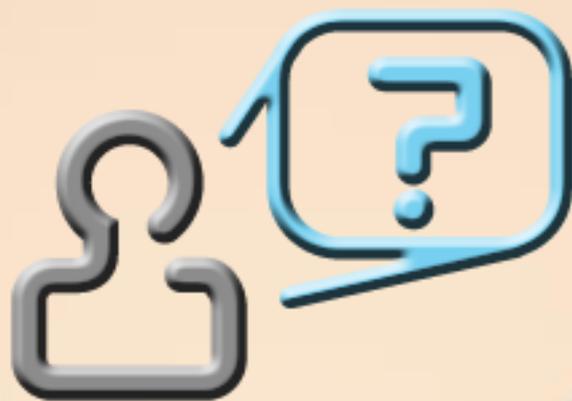
- 外観

```
public string SayFizzBuzz(int n)
```

- 反応

入力 (引数 n)		出力 (返値)
3で割り切れる	5で割り切れる	
YES	YES	"Fizz Buzz"
YES	NO	"Fizz"
NO	YES	"Buzz"
NO	NO	n.ToString()





今日やるのは…

内部設計（実装）を想定しないと
外部設計が上手く書けないパターン

例題: 「回文候補生」

- 入力文字列を前半として、回文の候補を表示する

かい - ぶん 【回文・廻文】

(カイモンとも)

(1)…略…

(2)和歌・連歌・俳諧などで、上から読んでも下から読んでも同音のもの。

(広辞苑 第四版 電子ブック版より)

- この心臓部 (候補文字列の生成) の外部設計をやろう



回答例(1): ドメインにも内部設計にも弱い人

- 外観 (シグネチャ)
string 回文候補を作る (string 文字列)
- 反応 (メッセージに対する応答)
引数「文字列」から回文を作成して返す

落第

致命的な問題点:

- ドメインを知らず、回文候補は1つだけだとしてしまった
- 回文を定義していない (← 検証不能)



回答例(2): ドメインは分かるが内部設計に弱い人

- 外観

string[2] 回文候補を作る(string 文字列)

- 反応

「文字列」の並び順を逆に入れ替えたものを「逆文字列」として、
返値[0]=「文字列」+「逆文字列」

返値[1]=「文字列」+「逆文字列の先頭1文字を削ったもの」

※ 「あい」を入れると、[0]は「あいいあ」、[1]は「あいいあ」

○ 回文の定義と例示がある

問題点:

-入力が空文字などのときは?

-いまどき配列? カンベンして…w



回答例(3): ドメインも内部設計も分かってる人

- 外観

`IList<string>` 回文候補を作る (string 文字列)

- 反応

「文字列」の並び順を逆に入れ替えたものを「逆文字列」として、
返値[0] = 「文字列」 + 「逆文字列」

返値[1] = 「文字列」 + 「逆文字列の先頭1文字を削ったもの」

※「あい」を入れると、[0]は「あいいあ」、[1]は「あいい」となる。
ただし、「文字列」が空文字のときは返値[0]のみ、「文字列」がnullのときは要素を持たないコレクションを返す。

○ 入出力の全パターンを網羅している

※長すぎる文字列は、.NET Framework ではメモリ不足で死ぬだけだから、考慮不要



回答例(3): 入出力表に書いてみる

入力文字列	出力 (返値)
null	要素無しのコレクション
空文字列	返値[0]=string.Empty のみ
1文字以上	返値[0]=「文字列」+「逆文字列」 返値[1]=「文字列」+「逆文字列の先頭1文字を削ったもの」

具体的なテストケースを考えてみよう

null と空文字列のケースは、すぐ書ける。

1文字以上のケースは、これだけで書けるだろうか？

このスペックでは具体性に欠けている



1文字以上の領域を、さらに分割

- 1文字以上の領域を、**内部設計を思い浮かべて**、さらに分割してしまおう!
- 内部で行う操作には、たぶんこんなものが:
文字列の結合、先頭文字の削除、文字列の逆転
- 必要になる操作の組み合わせでパターン分け

入出力表 (改)

入力文字列	出力 (返値)	
null	要素無しのコレクション	
空文字列	返値[0]=string.Empty のみ	
1文字以上	返値[0]=「文字列」 + 「逆文字列」 返値[1]=「文字列」 + 「逆文字列の先頭1文字を削ったもの」	
詳細パターン	例示	想定される内部操作
1文字	「あ」 → 「ああ」, 「あ」	文字列の結合
2文字以上 (回文)	「ああ」 → 「ああああ」, 「あああ」	文字列の結合 先頭文字の削除
2文字以上 (非回文)	「あい」 → 「あいいあ」, 「あいあ」	文字列の結合 先頭文字の削除 文字列の逆転

これならテストファーストできる♪ (テストコード)

```
[TestMethod]
public void 回文候補を作るTest01_null()
{
    var 候補List
        = 回文候補生ロジック.回文候補を作る(null);
    Assert.AreEqual(0, 候補List.Count);
}
```

```
[TestMethod]
public void 回文候補を作るTest02_空文字()
{
    var 候補List = 回文候補生ロジック
        .回文候補を作る(string.Empty);
    Assert.AreEqual(1, 候補List.Count);
    Assert.AreEqual(string.Empty, 候補List[0]);
}
```

```
[TestMethod]
public void 回文候補を作るTest03_あ()
{
    var 候補List
        = 回文候補生ロジック.回文候補を作る("あ");
    Assert.AreEqual("ああ", 候補List[0]);
    Assert.AreEqual("あ", 候補List[1]);
}
```

```
[TestMethod]
public void 回文候補を作るTest04_ああ()
{
    var 候補List
        = 回文候補生ロジック.回文候補を作る("ああ");
    Assert.AreEqual("ああああ", 候補List[0]);
    Assert.AreEqual("あああ", 候補List[1]);
}
```

```
[TestMethod]
public void 回文候補を作るTest05_あい()
{
    var 候補List
        = 回文候補生ロジック.回文候補を作る("あい");
    Assert.AreEqual("あいいあ", 候補List[0]);
    Assert.AreEqual("あいい", 候補List[1]);
}
```



これならテストファーストできる♪ (製品コード)

```
// リファクタ前の状態
public static IList<string>
    回文候補を作る (string src)
{
    if (src == null) //Test02
        //Test01
        return new List<string>();

    if (src == string.Empty) //Test03
    {
        //Test02
        var result = new List<string>();
        result.Add(string.Empty);
        return result;
    }

    //Test03
    //var 候補1 = src + src;
    //var 候補2 = src;

    //Test04
    //var 候補2 = src + src.Substring(1);

    //Test05
    var 逆文字列
        = new string(src.Reverse().ToArray());
    var 候補1 = src + 逆文字列;
    var 候補2 = src + 逆文字列.Substring(1);

    //Test03
    var 候補List = new List<string>();
    候補List.Add(候補1);
    候補List.Add(候補2);
    return 候補List;
}
```



ここまでのまとめ

- 前提: 外部設計は内部設計を縛るべきではない (←リファクタリングできなくなる)
- 内部設計を想定して外部設計を書くと具体性が増す
⇒ 内部設計(=実装)しやすい
- 外部設計で例示を増やしても、内部設計を縛ることにはならないので OK



演習: ソートの外部設計

- 仕様
 - 入出力は `IList<int>`
 - ※ ジェネリックに不慣れなら `int` の配列などでも可
 - 昇順にソートする
- 外観
 - `static IList<int> WankumaSort(IList<int>)`

```
static IList<int> WankumaSort(IList<int>)
```

整数を昇順にソートする



筆記用具のご用意を♪

演習タイム

ソートの外部設計



回答例

- 外観

static IList<int> WankumaSort(IList<int>)

ただし、引数と返値は同一インスタンス

- 応答

バブルソートを想定

入力数列	出力 (返値)	
null	null	
空リスト	引数と同一のインスタンス (以下同じ)	
1つ以上	昇順にソートされた数列	
詳細パターン	例示	想定される内部操作
交換無し	{1, 2} → {1, 2}	比較だけ一巡して終了
1巡だけ交換	{5, 3} → {3, 5}	比較と交換を一巡だけ行う
2巡以上交換	{7, 5, 3} → {3, 5, 7}	比較と交換を二巡以上 (交換不要になるまで) 行う





ご清聴ありがとうございました

