**～ 上手く TDD するために ～**

2009.9.12 biac
http://www.tdd-net.jp/

# 自己紹介

- 山本 康彦（ **biac** ）
  - いまだにプログラムを書きたがる 52歳
  - http://bluewatersoft.cocolog-nifty.com/
    ※ ハンドルで ぐぐってもらえば見つかる (経済産業諮問委員会 じゃないほう )
- 名古屋のとある ISV 勤務
  - この春まで、 WPF を使った業務アプリケーションの開発プロジェクトで品質保証を担当
  - MFS Agile を部分的に実施してみた
- **もとは機械の設計屋さん**
  - ものごとの見方・考え方が、きっとズレてる

# 宣伝：tdd-net.jp

TDD.NET: TDD とは? - Windows Internet Explorer
http://www.tdd-net.jp/whats-tdd.html

TDD.NET
.NET な開発者のための Test-Driven Development

コンテンツ

【記事】
・TDD とは？
・NUnit の使い方
├ NUnit 2.5 の導入 ...
└ C# 2008 Express + ...

【リンク集】
└ TDD.NET の道具箱

【blog】
* 下の [カテゴリー] と [バックナンバー] を参照してください
* [TDD の練習] 一覧

Tech・Ed 2009

tech·ed Japan 2009
2,000 人の IT エンジニアが集う
3 日限りの真夏の祭典

## TDD とは？

2009/06/29 biac

1. Test-Driven Development
2. TDD の概要
3. TDD の位置付け
4. TDD のメリット
5. TDD の効果
6. TDD の前提となるスキル

### 1. Test-Driven Development

TDD とは、ソフトウェア開発では Test-Driven Development のことです。
プログラムの仕様からソースコードを創り出す手順を規定しています。具体的には、テストファーストとリファクタリングを組み合わせた手法です。日本語では、一般的に「テスト駆動開発」と訳しています。

なお、検索をするときに "TDD" だけだと、通信技術の "Time Division Duplex" (時分割複信) や、仮想の強襲揚陸潜水艦の名前 "Tuatha De Danann" が引っ掛かってきますので、"テスト" とか "プログラム" といった単語も合わせて検索するとよいです。

### 2. TDD の概要

RSSを表示する

プロフィール

最近のトラックバック

最近のコメント

biac on [お知らせ] ようこそ、TDD.NET へ

biac on [お知らせ] 「TDD.NET の道具箱」にリンクを追加

まさる on [お知らせ] 「TDD.NET の道具箱」にリンクを追加

D on [お知らせ] ようこそ、TDD.NET へ

biac on [お知らせ] ようこそ、TDD.NET へ

ページが表示されました　インターネット | 保護モード: 有効　100%

# Tech・Ed 2009 横浜に行ってきました

- …初日だけ f(^^;
- BoF-02: Visual Studio 2010 で進化するテスト環境
  - えムナウ、επιστημη、他1名
- T2-305: Silverlight 3 の新機能 by MSKK 大西 彰
- LT-01: TDD とメソッドの外部設計 by biac
  - LT 登壇者7名のうち、3名が わんくま だったらしい

写真撮影: 原水 真一 (MSKK)

写真撮影: 原水 真一 (MSKK)

# アジェンダ

- TDD のおさらい と **やってみると難しい** ということ
- **メソッドの外部設計**をやろう ということ
- Visual Studio 2010 で **TDD のための機能**がさらに強化されている ということ

# Test Driven Development

- TDD = テストファースト + リファクタリング

- 1. テストコードを書く。 (RED)
  2. テストに通る製品コードを書く。 (GREEN)
  3. リファクタリングする。
  → 1. に戻る

- 1.～2. がテストファースト
  ※ これが出来ないと TDD にならない

# テストファーストの効果

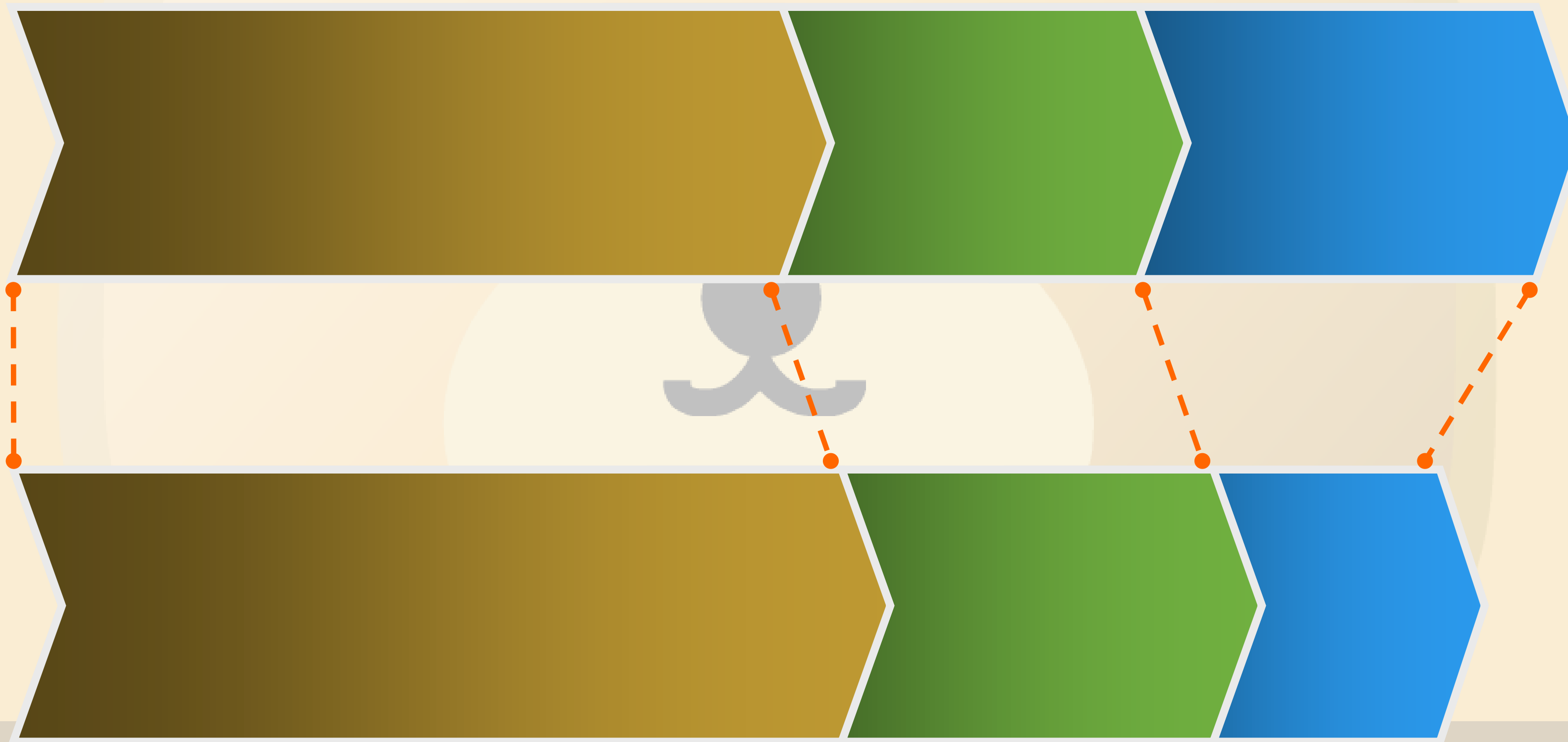
- 品質保証的に…
- 品質向上 (バグ減)
  - 設計書レビュー効果
  - 単体テスト実施効果
  - それぞれで、バグが 3割以上減少<br>0.7 × 0.7 ⇒ 半分以下になる！(結合テスト 2回分)
- 結合テストの半分以上は**バグ対応**<br>バグレポート・トリアージ・修正・確認テスト<br>⇒ この**工数が半分以下に !!**

# テストファーストの効果

実装

結合テスト
テスト実施

結合テスト
バグ対応



# TDD の効用

- 開発者的に…
  - **安心**
    - いつでもテストを実施して、 壊していないことを確認できる
    - ユニットテストを書き始めたら、 目の前のメソッドだけに集中できる。 悩まなくていい。
  - **楽しい**

    好きなだけ (時間さえ許せば)、 リファクタリングできる

※ 機械設計屋さん的には…

テストケース ( テスト方法と合格判定値 ) 無しでは、 設計しようがないよぉ～ (; ;

## いいことずくめの TDD …、

# ところが！

実際にやってみると…

# ユニットテストを上手く書けない!!

- なにを書けばいいか、わからない！
- テストケースが足りない！
- 無駄なユニットテストを書いてしまう！

⇒ 原因は？

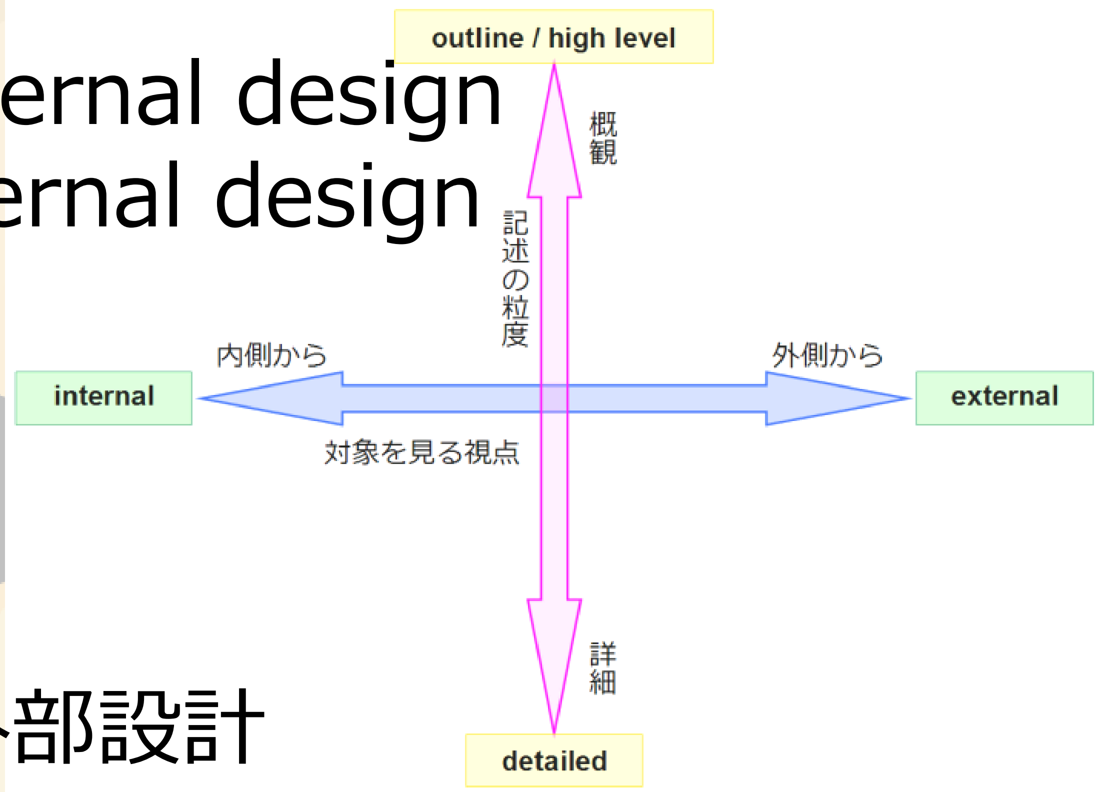
- いろいろ聞いてみると、　どうやら…
  **メソッドの外部設計が上手く出来ない!!**

# アジェンダ

- TDD のおさらい と **やってみると難しい** ということ
- **メソッドの外部設計**をやろう ということ
- Visual Studio 2010 で **TDD のための機能**がさらに強化されている ということ

# メソッドの設計

- 外部設計 external design
  内部設計 internal design




- メソッドの外部設計
  - 静的: シグネチャ (引数/返値)
  - 動的: ふるまい (入力/出力)

インターフェース

# メソッドのふるまいを定義する

- メソッドのふるまい (入出力) を定義するには、どうするか?
  - メソッドのふるまいに対して**影響を及ぼすもの** (入力) をすべて見つけ出す。
    引数、メンバー変数、中から呼び出したメソッドの返値… etc.
  - メソッドのふるまいによって**影響を受けるもの** (出力) をすべて見つけ出す。
    返値、メンバー変数、呼び出したメソッドで影響されるもの…
  - 入出力の**組み合わせパターンをすべて**定義する。

# 外部設計の例 ～ 単純なメソッド

- 1入力 – 1出力

string BuildMessage(string targetName)

文字列 {foo} から、“Hello, {foo} !” という文字列を作り出す。

| **入力**<br>string targetName | **出力**<br>返値 string |
| --- | --- |
| null | (NullReferenceException) |
| "" (空文字) | "Hello !!" |
| "{foo}" (1文字以上) | "Hello, {foo} !" |

http://bluewatersoft.cocolog-nifty.com/blog/2009/05/1-1f16.html

# ユニットテストとして書き下す

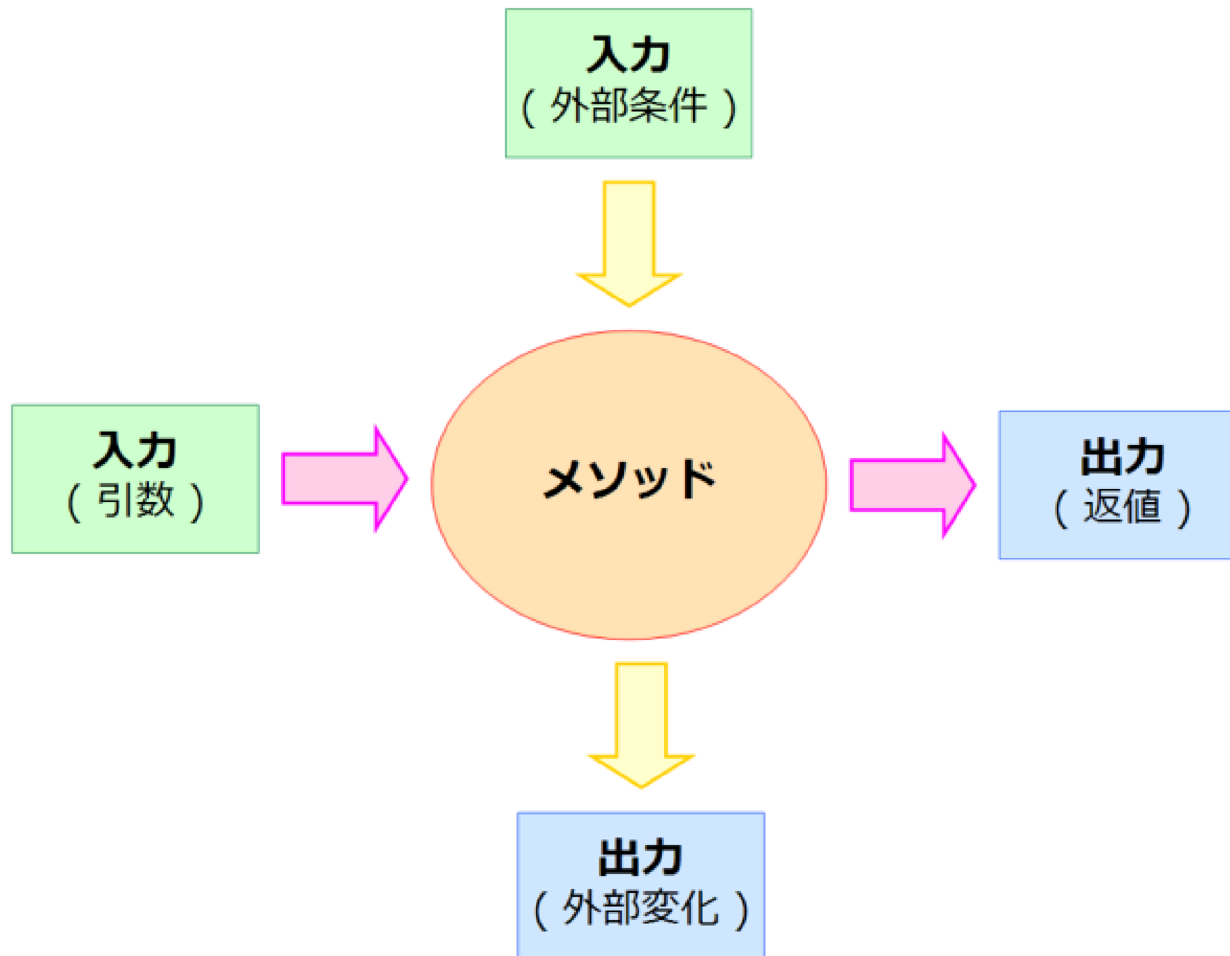
- 入出力表の各行が、ひとつのテスト

```
[TestMethod]
[ExpectedException(typeof(NullReferenceException))]
public void BuildMessageTest_nullを渡す() {
  Greeter g = new Greeter();
  string dummyResult = g.BuildMessage((string)null);
  Assert.Fail("期待した例外が発生しませんでした。");
}

[TestMethod]
public void BuildMessageTest_空文字を渡す() {
  Greeter g = new Greeter();
  Assert.AreEqual("Hello !!", g.BuildMessage(string.Empty));
}

[TestMethod]
public void BuildMessageTest_1文字以上の文字列を渡す() {
  Greeter g = new Greeter();
  Assert.AreEqual("Hello, NoMan !", g.BuildMessage("NoMan"));
}
```

# 外部設計の例 ～ 複雑な入出力



# 外部設計の例 ～ 複雑な入出力

string BuildMessageAndSetAmPm(string targetName)

・文字列 {foo} から、“Hello, {foo} !” という文字列を作り出す。
・また、メンバ変数 AmPm に午前/午後の区別を書き込む。

※ ただし、targetName が空文字のときは “Hello !!” を返す。
※ ただし、"Hello" の部分は、朝 (5時～10時) は "Good morning"、昼 (10時～18時) は "Hello"、夕方 (18時～20時) は "Good evening"、それ以降は "Good night" とする。

- 入力 ～ 引数 targetName と、システム時刻
  3パターン × 6パターン ⇒ 18パターン?
- 出力 ～ string の返値と、 メンバー変数 AmPm

http://bluewatersoft.cocolog-nifty.com/blog/2009/05/2-8801.html

# 外部設計の例 ～ 複雑な入出力

| 入力 | | 出力 | |
|---|---|---|---|
| string targetName | システム時刻 t | メンバ変数 AmPm | 返値 string |
| null | 0:00 <= t < 12:00 | 午前 | (NullReferenceException) |
| null | 12:00 <= t < 24:00 | 午後 | (NullReferenceException) |
| "" (空文字) | 0:00 <= t < 5:00 | 午前 | "Good night !!" |
| "" (空文字) | 5:00 <= t < 10:00 | 午前 | "Good morning !!" |
| "" (空文字) | 10:00 <= t < 12:00 | 午前 | "Hello !!" |
| "" (空文字) | 12:00 <= t < 18:00 | 午後 | "Hello !!" |
| "" (空文字) | 18:00 <= t < 20:00 | 午後 | "Good evening !!" |
| "" (空文字) | 20:00 <= t < 24:00 | 午後 | "Good night !!" |
| "{foo}" (1文字以上) | 0:00 <= t < 5:00 | 午前 | "Good night, {foo} !" |
| "{foo}" (1文字以上) | 5:00 <= t < 10:00 | 午前 | "Good morning, {foo} !" |
| "{foo}" (1文字以上) | 10:00 <= t < 12:00 | 午前 | "Hello, {foo} !" |
| "{foo}" (1文字以上) | 12:00 <= t < 18:00 | 午後 | "Hello, {foo} !" |
| "{foo}" (1文字以上) | 18:00 <= t < 20:00 | 午後 | "Good evening, {foo} !" |
| "{foo}" (1文字以上) | 20:00 <= t < 24:00 | 午後 | "Good night, {foo} !" |

# 組み合わせの爆発

- 前の例でも 14通りになった
  ⇒ 入力がもっと増えたらどうなる?
  **テストケース数の爆発!!**

- 対処は?
  ⇒ メソッドを分割する

例えば、「時刻を渡すと、メンバー変数 AmPm に午前/午後をセットする」メソッド SetAmPm() を切り出したら?
例えば、「時刻を渡すと、挨拶 (“Hello” とか “Good morning” とか) を返してくれる」メソッドを切り出したら?

# メソッド分割で、組み合わせ爆発を防ぐ

- string **GetGreet**(DateTime t)

| **入力**<br>DateTime t | **出力**<br>返値 string |
|---|---|
| 0:00 <= t < 5:00 | "Good night" |
| 5:00 <= t < 10:00 | "Good morning" |
| 10:00 <= t < 18:00 | "Hello" |
| 18:00 <= t < 20:00 | "Good evening" |
| 20:00 <= t < 24:00 | "Good night" |

- void **SetAmPm**(DateTime t)

| **入力**<br>DateTime t | **出力**<br>メンバー変数 AmPm |
|---|---|
| 0:00 <= t < 12:00 | 午前 |
| 12:00 <= t < 24:00 | 午後 |

- string **BuildMessageAndSetAmPm**(string targetName)

| 入力 | | 出力 | |
|---|---|---|---|
| string targetName | GetGreet(DateTime .Now) の返値 | メンバ変数 AmPm | 返値 string |
| null | "{bar}" (1文字以上) | SetAmPm() 呼び出し | (NullReferenceException) |
| "" (空文字) | "{bar}" (1文字以上) | SetAmPm() 呼び出し | "{bar} !!" |
| "{foo}" (1文字以上) | "{bar}" (1文字以上) | SetAmPm() 呼び出し | "{bar}, {foo} !" |

- 元は 14パターン ⇒ トータルで 10パターン、 個々の表は 2～5パターンに減らすことができた。

# メソッドの外部設計をしよう

- テストファーストに慣れるまでは、ふるまいを定義する入出力表を書こう
- 慣れてきたら、表を書かなくてもテストコードを書けるようになる
- さらに慣れてきたら、「TDD 三原則」

# TDD 三原則

- Robert C. Martin (UncleBob)
  - 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。

http://www.tdd-net.jp/2009/08/tdd-9534.html

# アジェンダ

- TDD のおさらい と **やってみると難しい** ということ
- **メソッドの外部設計**をやろう ということ
- Visual Studio 2010 で **TDD のための機能**がさらに強化されている ということ

# VS2010 の TDD 向け新機能

- クラスやメソッドのスケルトンを自動生成
  "generate from usage"
- GUI の自動テスト
  “Coded UI Test”
- (おまけ) Quick Search の camel-case match
- TFS (未確認)
  - “Test Lab Manger”
  - テスト影響分析 “Test Impact View”
  - “Gated Check-in” (チェックインされるとビルド処理が作動)
  - ワークフローベースのビルドエンジン