Visual Studio 2008 でやる テスト駆動開発

2008/04/26 biac
http://bluewatersoft.cocolog-nifty.com/
機材協力: 日本インフォメーション㈱

自己紹介

- 山本 康彦 (**biac**)
  - いまだにプログラムを書きたがる 50歳
    http://bluewatersoft.cocolog-nifty.com/blog/cat8051143/
- 名古屋のとある ISV 勤務
  - 現在、 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中
    http://www.nicnet.co.jp/page/d_system/d03_12_jisseki.html
  - MSF Agile を部分的に実施中
- **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

テスト駆動開発 (TDD)

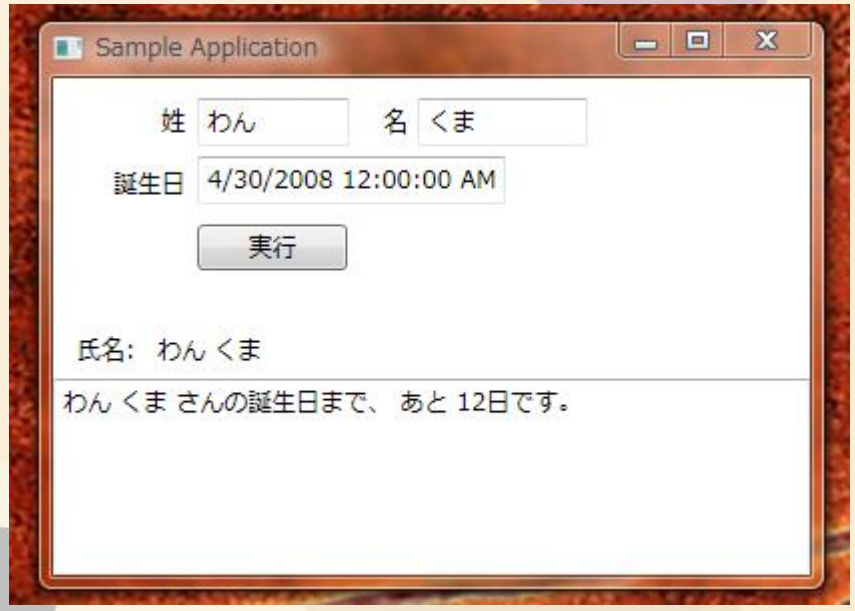
- Test Driven Development
  - 1. 単体テストを書く
    テストは失敗する **(RED)**
  - 2. 製品コードを書く
    テストを成功させる **(GREEN)**
  - 3. リファクタリング **(REFACTOR)**
    コードを綺麗に
    → 1. に戻る
- **RED** → **GREEN** → **REFACTOR** の
  リズムに乗って、 さくさく開発

Visual Studio 2008

- **Professional** Edition
  - 単体テスト機能はアリ
    http://msdn2.microsoft.com/ja-jp/library/bb385902.aspx
- **Team System** Development Edition
  - さらに、テストカバレッジ、コード分析などの機能
    http://msdn2.microsoft.com/ja-jp/library/47f7hz7y.aspx
- **Express / Std は？**
  - オープンソースのツールを利用しよう!
    - NUnit, NCover, FxCop …

こんなものを作ってみよう

- どんなもの?
  - 誕生日までの日数を答えてくれる。
- どんな構造?
  - WPF の画面
  - 画面を抽象化したモデル
  - 返答を作るロジック
- 単体テストできるのは?
  - ロジック部分のみ
  - UI は、 まだ難しい



・ UI とロジックを分離すべし
・ UI のコード量は減らすべし

DEMO 0

DEMO

画面 / UIModel / Logic の スケルトン まで作っておく

最初の一歩

- 1. [製品] スケルトンを書く
  - メソッドのシグネチャと、 仮の return 文を書く。
  - TDD の本来の流儀からは、 外れている。
- 2. 単体テストを生成する
  - 「単体テストウィザード」を使って、 単体テストコードを自動生成する。
- 3. [テスト] テストコードを書く
- 4. テストをする (失敗するはず)
- 5. [製品] テストを通るだけのコードを書く
- 6. テストをする (成功するまで)

DEMO

最初のテスト と コードカバレッジや コード分析

コードの品質保証

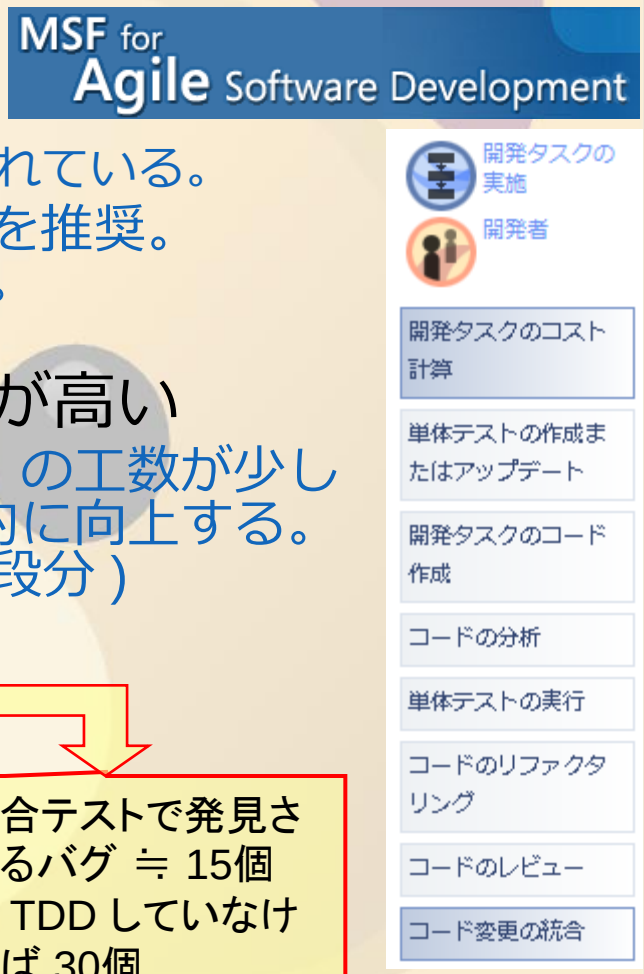
- ※ 以下は、Team System の機能
- テストカバレッジ
  - 単体テストのときに、 実行されたコード行をカウント。 ソースを色分けして表示。 (C0 カバレッジ)
    http://msdn2.microsoft.com/ja-jp/library/y8hcsad3.aspx
  - カバーされていないコード == 不要なコード (?)
  - オープンソース: NCover & NCoverExplorer
- コード分析
  - Microsoft の基準に沿ったコーディングをしているか、 チェック。
    http://msdn2.microsoft.com/ja-jp/library/y8hcsad3.aspx
  - 無視することもできるけど、 後で泣くのは誰?
  - MS の無償ツール: FxCop
- コードメトリックス
  - コードの複雑さや保守性を測定する。
    http://msdn2.microsoft.com/ja-jp/library/bb385910.aspx
  - 点数が悪すぎるところは、 リファクタリング候補。

なぜ単体テストコード?

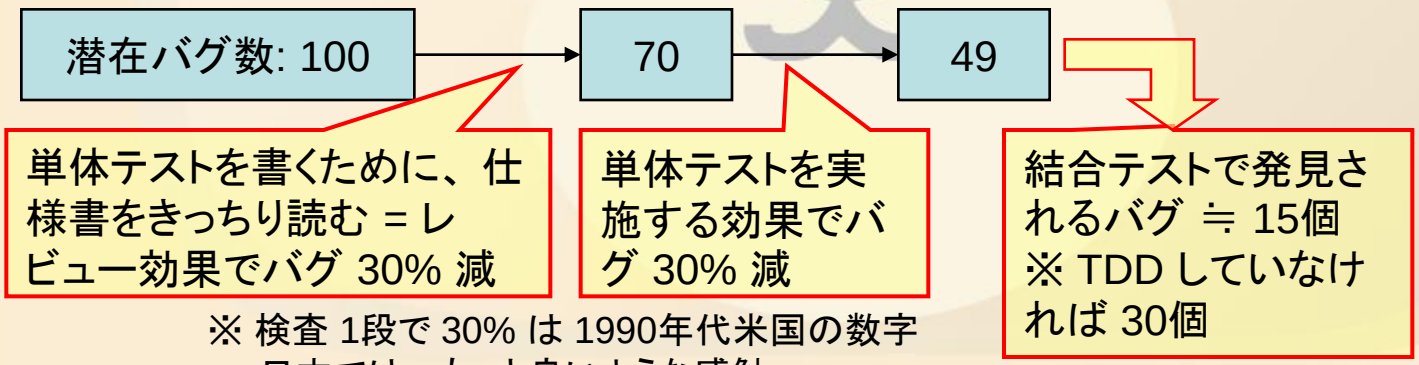
- 単体テスト == メソッドレベルの外部設計書
  - メソッドのシグネチャと戻り値の定義 (外部設計) は、昔は書いていた (…らしい)。
  - 厳密に書くのは、自然言語ではムリ。
  - コードを書けない SE の書いた設計書では、たいがい作れない。
- テストに通る == ゴール
  - 自分のリズムに合ったゴールを設定し、さくさく開発。
  - テストコードを書いた時点で、 わかりにくい仕様書のことは、とりあえず忘れて OK !
- 自動実行できる == リファクタリングできる
  - いつでもすぐに単体テストを実行して、 コードを壊していないことを証明できる。時間さえ許すなら、 気が済むまでコードを改良できる。

TDD は今や常識

- 開発プロセスの一部
  - たとえば MSF Agile Ver.4
    - 「開発タスク」に TDD が取り入れられている。
  - そのほかの Agile プロセスも、 TDD を推奨。
    TDD を排斥しているプロセスは無い。



- 品質向上のコストパフォーマンスが高い
  - 開発プロセス全体では、「実装工程」の工数が少し増えるだけで、コードの品質は飛躍的に向上する。
    (同じ品質にするなら、結合テスト 2段分)



※ 検査 1段で 30% は 1990年代米国の数字
日本では、もっと良いような感触。

テストメソッドの基本 (1)

- テストの起承転結

```
[TestMethod]
public void FooMethodTest()
{
  // テストの準備（不要なこともある）
  //   → 共通部分は TestInitialize, ClassInitialize へ

  // テスト実行 --- 製品コードの呼び出し
  string answer = BarClass.FooMethod();

  // 結果判定
  Assert.AreEqual("Hello, TDD!", answer);

  // テストの後始末（不要なこともある）
  //   → 共通部分は TestCleanup, ClassCleanup へ
}
```

簡単な場合は、実行と判定を同一行にまとめてもよい。

テストメソッドの基本 (2)

- テストケースごとにテストメソッド

```
[TestMethod]
public void FooMethodTestcase1Test(){ ・・・ }

[TestMethod]
public void FooMethodTestcase2Test(){ ・・・ }
```

- 一連のテストケースでひとつのテストメソッド

```
[TestMethod]
public void FooMethodTest(){
  // Testcase1
  { ・・・ }

  // Testcase2
  { ・・・ }
}
```

プロジェクト内でどちらかに統一されていれば良いと思う。

テストメソッドの基本 (3)

- 例外を期待するテスト: ExpectedException 属性

```
[TestMethod]
[ExpectedException(typeof(System.DivideByZeroException))]
public void FooMethodTest(){
  int n = BarClass.FooMethod(0); // 0除算例外が期待される
}
```

- 例外を期待するテスト: try ～ catch

```
[TestMethod]
public void FooMethodTest(){
  try {
    int n = BarClass.FooMethod(0); // 0除算例外が期待される
    Assert.Fail("例外が出て、ここには来ないはずです。");
  }
  catch (System.DivideByZeroException){
    // ( success! )
  }
}
```

テストコードで catch するのは、このパターンだけ。
リソース解放が必要なら finally 句で。

VS便利機能(1) private メソッドのテスト

- NUnit では、 private メソッドはテストできない
  - public メソッド経由でテストされているはず。
  - 複雑な private メソッドなので、どうしてもそこだけテストしたいときは… public にしてしまったり、 #if DEBUG してみたり。
- プライベート アクセッサ
  - 単体テストウィザードで、 private メソッドを指定して単体テストを作ると、 自動的にプライベートアクセッサが生成される。
    http://msdn2.microsoft.com/ja-jp/library/ms184807.aspx
- InternalsVisibleTo 属性
  - 単体テストウィザードで [InternalsVisibleTo 属性を追加する] と、internal メソッドが見えるようになる。
    http://msdn2.microsoft.com/ja-jp/library/bb385840.aspx

DEMO 2

DEMO

private メソッドのテスト

VS便利機能(2) データドリブン単体テスト

- 条件を変えて同じテストをする
  - 引数や予想される結果が違うだけで、 同じテストロジックを何度も書くことはめんどくさい。
- データドリブン単体テスト
  - DataSource 属性を付けることで、 データソースから1行読み込むごとにテストメソッドが 1回呼び出されるようになる。
    http://msdn2.microsoft.com/ja-jp/library/ms182528.aspx
  - 読み込まれたレコードには、TestContext.DataRow[“{列名}”] でアクセス。
  - データソースには、データベース、 CSV ファイル、XML ファイルが利用可能
    ODBC データソースが使える == Excel ファイルも OK

DEMO

Excel を使った
データドリブンテスト

TDD を始めよう

- TDD は楽しい
  - コードを書く時間が増える。(あいまいな仕様書のことで悩んでいる時間が減る。 テストが書けないなら、 仕様書がおかしい！)
  - リファクタリングできる !!
- 参考書
  - 「**Microsoft.NET でのテスト駆動開発**」(ジェームス・ニューカーク著) ISBN-13: 978-4891004422
    http://www.amazon.co.jp/exec/obidos/ASIN/4891004428/bluewatersoft-22
  - 「**リファクタリング** ― プログラムの体質改善テクニック」(マーチン ファウラー著) ISBN-13: 978-4894712287
    http://www.amazon.co.jp/exec/obidos/ASIN/4894712288/bluewatersoft-22
  - Guidelines for Test-Driven Development
    http://msdn2.microsoft.com/en-us/library/aa730844.aspx
- このスライドとサンプルコード
  - 次の場所に置いてあります。
    http://bluewatersoft.cocolog-nifty.com/blog/cat8051143/

ありがとうございました