2008/7/26

タダで始めるテストファースト入門

C# Express + NUnit

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

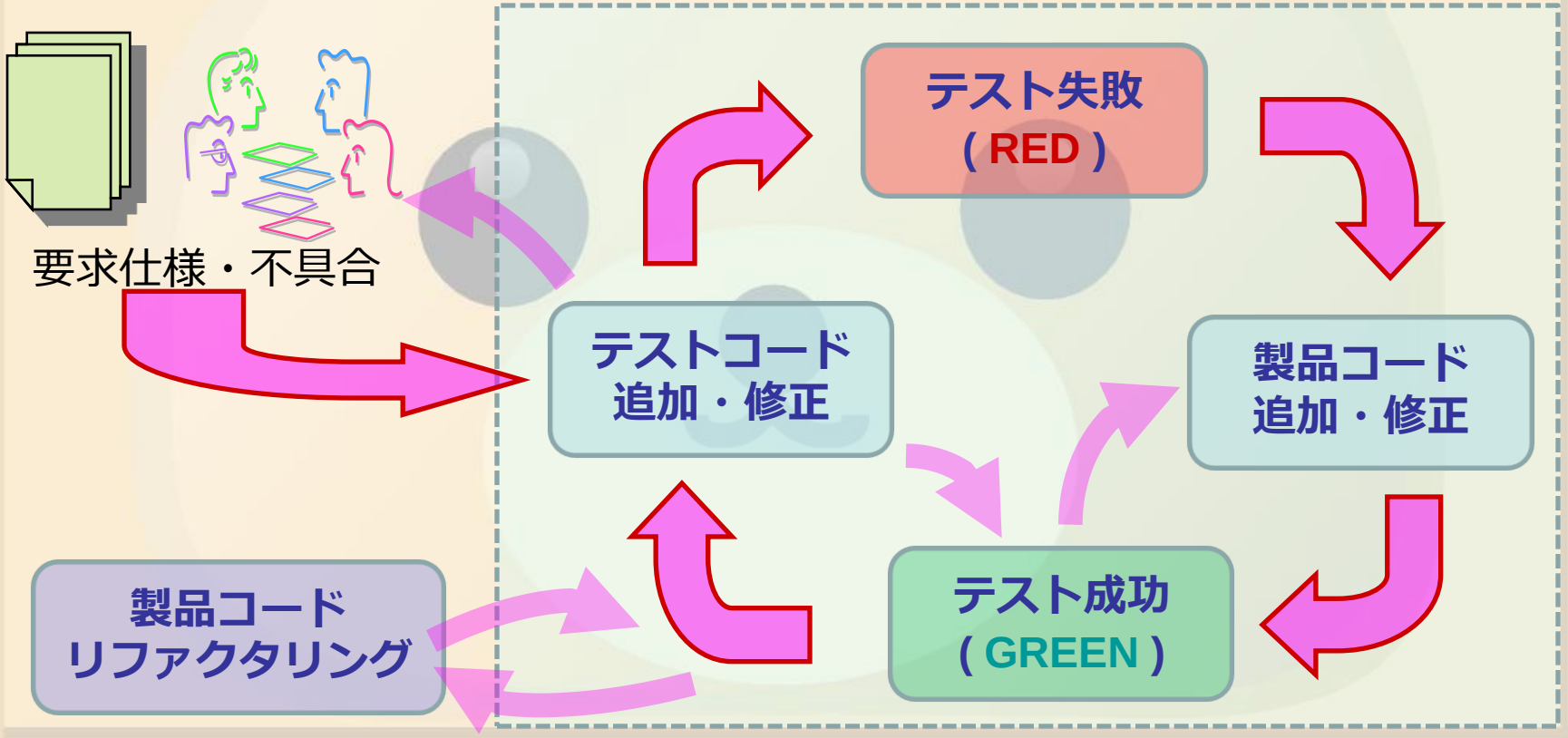
- 山本 康彦 (**biac**)
  - いまだにプログラムを書きたがる 51歳
  - http://bluewatersoft.cocolog-nifty.com/blog/cat8051143/
- 名古屋のとある ISV 勤務
  - 現在、 WPF を使った業務アプリケーションの開発プロジェクトで品質保証を担当
  - MFS Agile を部分的に実施中
- **もとは機械の設計屋さん**
  - ものごとの見方・考え方が、きっとズレてる

用意するもの

- 開発環境とテストフレームワーク
  - C# 2008 Express
    http://www.microsoft.com/japan/msdn/vstudio/Express/
  - NUnit 2.4.7
    http://www.nunit.org/index.php
- あるといいなぁ
  - 静的コード分析ツール FxCop 1.35
  - テストカバレッジ計測ツール PartCover

テスト駆動開発 と テストファースト

- この図の全体がテスト駆動開発 (TDD) で、テストファースト技法はその一部。 (点線内)



最初のテスト

- ソリューションとテストプロジェクトを作る。
- テストプロジェクトに参照設定を追加
  nunit.framework
- テスト: NUnit Framework をちゃんと呼び出すことができるか?

最初のテスト(コード)

```
using NUnit.Framework;

namespace Wankuma.SorterSampleTest {

        [TestFixture]
        public class WankumaSorterTest {

                [Test]
                [Description("初めての NUnit テスト")]
                public void FirstTest() {
                    Assert.Fail("ちゃんと Assert できました! (^^;");
                }
        }
}
```

-------- src00.sln --------

- このテストは失敗します(レッド)
  失敗させようとして、 失敗した → ちゃんと動いている!

ほんとの 入門編

ソートしてくれるクラスを作ろう

- 仕様
  - 入出力は int の配列
  - とりあえず、 昇順にソート
- 最初に考えること – どんなふうに使う?
  - クラス名、 メソッドシグネチャを決める

```
int[] input = ・・・

WankumaSorter sorter = new WankumaSorter();
int[] result = sorter.Sort(input);
```

テストケース: n = 1

- 最初は、 仕様の一番簡単なところから。
  n = 1 のとき
  入力: int[] input = { 7 } // 数字 1つ
  結果: int[] result = { 7 }

  ※ 簡単なところから、 テストをちょこっと、製品をちょこっと…
  考えなきゃいけないことのスコープを小さくして、 シンプルに進めていく。

テストケース: n = 1 (テストコード)

```
[Test]
[Description("n=1 のとき (何もしない)")]
public void SortTestWith1Item() {
        // テストの準備
        int[] input = { 7 };

        // テスト実行
        WankumaSorter sorter = new WankumaSorter();
        int[] result = sorter.Sort(input);

        // テスト結果の検証
        Assert.AreEqual(1, result.Length);
        Assert.AreEqual(7, result[0]);

        // 使ったリソースの後始末 ... 今回は無し
}
```

※ 検証用比較データ (上では 1 とか 7) は、 テスト対象とは独立して生成したものであること。

テストケース: n = 1 (製品コード)

- テストを通るギリギリのコードで製品を実装していく。

1. まず、 コンパイルが通るように。 製品のプロジェクトと WankumaSorter クラスを作る。

```
namespace Wankuma.SorterSample {
	public class WankumaSorter {
		public int[] Sort(int[] input) {
			return null; // ← とりあえず何か返す
		}
	}
}
```

- 2. テスト実行 --- 失敗します (RED)
  - まだ実装のまともな中身が無いんですから、失敗しなきゃいけないですね。 ちゃんと失敗することを確認します。
  - 失敗するはずだと思っていて、 万一成功してしまったら… なにか見落としていることがあるはずですね。

- 3. 製品の実装 --- テストに通る最小限度
  - このテストを通すには、{7}を返してやればいいんです。(シンプルに!)
- 4. テスト実行 --- 成功します(GREEN!)

```
namespace Wankuma.SorterSample {
        public class WankumaSorter {
                public int[] Sort(int[] input) {
                        return new int[]{7};
                }
        }
}
```

-------- src01.sln --------

宿題 – その1

- 最初に決めておくべきことは、これでOK?

※ 帰ってくるのは同じオブジェクト? それとも?
→ Assert.AreSame() または Assert.AreNotSame() を使ってテスト

テストケース: n = 2

- n = 2 のとき
  入力: int[] input = { 3, 5 } // 数字 2つ
  結果: int[] result = { 3, 5 }

  …これは、 何もせずそのまま返せば OK になっちゃうと思いますよね?
  簡単すぎなので、 パス。

  入力: int[] input = { 5, 3 } // 数字 2つ
  結果: int[] result = { 3, 5 }
  …これなら、 ちょっと製品の実装が進みそう。

テストケース: n = 2 (テストコード)

```
[Test]
[Description("n=2 のとき。 逆順なら入れ替え。")]
public void SortTestWith2Items() {
        int[] input = { 5, 3 };

        WankumaSorter sorter = new WankumaSorter();
        int[] result = sorter.Sort(input);

        CollectionAssert.AreEqual(
                        new int[] { 3, 5 }, result
        );
}
```

※ ちゃんとテストは失敗しますね?
　これから実装することに意味がある、 というものです

テストケース: n = 2 (製品コード)

- こんな実装でどうでしょう?

```
public int[] Sort(int[] input) {
        //return new int[]{7};
        if (input[0] > input[1]) {
                int work = input[0];
                input[0] = input[1];
                input[1] = work;
        }
        return input;
}
```

- さっき作ったテスト SortTestWith2Items() を流すと ...グリーン!
  これでOK?

- テストを全部流してみましょう ...レッド!? **orz**

  なぜですか?
  n = 1 のときも、 2つあると思って比較しちゃうからですね。
  Sort() の最初に、 こんなのを付け加えて

```
if (input.Length == 1)
    return input;
```

  テストは...? はい、オールグリーン!

  ※ いいかげんな実装だと思うでしょう。
  しかし、 入力される配列要素数が 1 または 2 である、 という前提が正しいなら、 これで製品として OK なんですよ。
  public メソッドですから、 ガード句

```
if (input.Length > 2)
    throw ArgumentOutOfRangeException();
```

  なんてのをメソッドの先頭に加えれば完璧。

ちょっとリファクタリング

- 製品コードの

```
int work = input[0];
input[0] = input[1];
input[1] = work;
```

の部分は、きっとこれからも使うでしょう。
この 3行で、値を入れ替えるという操作をやっています。
リファクタリングして、値を入れ替えるというメソッドにしておきましょう。
※ Express でも、メソッドの抽出リファクタリングをサポートする機能は付いています。

```
            if (input[0] > input[1]) {
                Swap(ref input[0], ref input[1]);
            }

    private static void Swap(ref int n, ref int m) {
            int work = n;
            n = m;
            m = work;
    }
```

- リファクタリングしたら、 また全部のテストを流して、 なにも壊していないことを確認しておきましょう。

※ テストもリファクタリングしましょうか。
テストの実行と結果の検証を区別するため、

```
int[] result = sorter.Sort(input);
CollectionAssert.AreEqual(new int[] { 3, 5 }, result);
```

などと書いてきましたけど。
まぎらわしくなければ、

```
CollectionAssert.AreEqual(new int[] { 3, 5 },
sorter.Sort(input));
```

と 1行にまとめて書いちゃっても同じですよね。

-------- src02.sln --------

テストケース: n = 3

- n = 3 のとき
  入力: int[] input = { 3, 7, 5 } // 数字 3つ
  結果: int[] result = { 3, 5, 7 }

  …これは失敗するでしょう。

※ どんなテストをどんな順序で作っていくと、効率よく製品コードが育っていくか…
これは、 テストファーストで作っていくときの醍醐味であり、 最も難しいところでもあります。

テストケース: n = 3 (テストコード)

```
[Test]
[Description("n=3 のとき。")]
public void SortTestWith3Items() {
        int[] input = { 3, 7, 5 };

        WankumaSorter sorter = new WankumaSorter();
        CollectionAssert.AreEqual(
                new int[] { 3, 5, 7 }, sorter.Sort(input)
        );
}
```

※レッドを確認したら、 製品コードを変えます。
こんどは、配列の 2つめと 3つめも比較して、 逆順だったら入れ替えるロジックを追加しましょうか?

いや、 もう先が見えてますよね?
4つになったら、 また同じことが...

テストケース: n = 3 (製品コード)

そこで、 for 文で書き直すことにします。

```
public int[] Sort(int[] input) {
        if (input.Length == 1)
                return input;

        for (int i = 0; i < (input.Length - 1); i++) {
                if (input[i] > input[i+1]) {
                        Swap(ref input[i], ref input[i+1]);
                }
        }
        return input;
}
```

テストケース: n = 3 (テストコード)

・・・はい、 オールグリーン!

これで OK?
ちょっとテストがヌルイような気がしませんか?

n = 3 のとき
入力: int[] input = { 7, 5, 3 } // 数字 3つ
結果: int[] result = { 3, 5, 7 }

これはどうでしょう?
テストを追加して、 走らせてみると… レッド!?

```
Wankuma.SorterSampleTest.WankumaSorterTest.SortTestWith3Items:
    Expected and actual are both <System.Int32[3]>
    Values differ at index [0]
    Expected: 3
    But was:  5
```

- 配列の先頭が 3 になっていて欲しいのに、 帰ってきたのは 5 です。 なぜでしょう?
  隣同士の交換を、 先頭から見て行って一回ずつしかやってないからですね。
  7, 5, 3
  ↓
  0番と 1番を比較・交換
  5, 7, 3
  ↓
  1番と 2番を比較・交換
  5, 3, 7 ←いまココ

テストケース: n = 3 (製品コード)

- for ループ 1回で、一番大きな数字が一番後ろに来ました。 そこは確定でしょう。
- けれど、 そこより前はまだ大小関係が入り乱れています。
- 外側に for ループを追加して、 内側の for ループの終了位置を一つずつ減らしながら、ループを繰り返してあげる必要があるみたいです。

```
for (int j = input.Length - 1; j > 0; j--) {
        for (int i = 0; i < j; i++) {
                if (input[i] > input[i + 1]) {
                        Swap(ref input[i], ref input[i + 1]);
                }
        }
}
```

- さぁ、これで... オールグリーン!

- いくつか確認のためのテストケースを追加してみましょう。（宿題その2）
  それらは、 最初からグリーンになるはずのテストです。

-------- src03.sln --------

ちょびッと 応用編

ムダな子はいねぇが～!?

- 結局、 ソートのロジックは二重ループになりました。
  ちょっといじわるなテストを考えてみます。
- n = 5 のとき
  入力配列: 1, 2, 3, 5, 7 (数字 5つ)
  結果配列: 1, 2, 3, 5, 7
- このとき、 Sort() の中で、 大小比較が何回行われるでしょう?
  頭から比較していって最後まで 4回比較してみれば、 すでに整列していることが分かります。
  4回より多く比較するのはムダってものです。

またちょびッとリファクタリング

- そこで、 比較回数を計測できるようにリファクタリングしてから、プローブを突っ込みます。
- リファクタリング - 比較部分をメソッドに切り出す

```
            if (IsNotInOrder(input[i], input[i + 1])) {
                    Swap(ref input[i], ref input[i + 1]);
            }

    private static bool IsNotInOrder(int lead, int trail ) {
            return (lead > trail);
    }
```

テストプローブの挿入 (製品コード)

- オールグリーンを確認したら、プローブを仕込みます。

```
#if DEBUG
        public static int TestCompareCount;
#endif

        private static bool IsNotInOrder(int lead, int trail ) {
#if DEBUG
                TestCompareCount++;
#endif
                return (lead > trail);
        }
```

テストケース: 最小の比較回数 (テストコード)

- すると、 比較は 4回だけにしてほしい、というテストが書けます。

```
[Test]
[Description("n=5 のとき。 最初から整列していると、比較は 4回。")]
public void SortTestWith5Items() {
#if DEBUG // ←プローブは DEBUG 時のみ有効なので、 テストも。
        int[] input = { 1, 2, 3, 5, 7 };

        WankumaSorter sorter = new WankumaSorter();
        sorter.TestCompareCount = 0;

        CollectionAssert.AreEqual(new int[] { 1, 2, 3, 5, 7 },
sorter.Sort(input)); // ←念のためソートできてることを確認
        Assert.AreEqual(4, sorter.TestCompareCount);
#endif
}
```

- テストしてみましょう... レッドです。

```
Wankuma.SorterSampleTest.WankumaSorterTest.SortTestWith5Items:
  Expected: 4
  But was:  10
```

- なんと 10回も比較しています。
  これはどうしたものでしょう...?

  ひとつの手として、 Swap() した回数を数えておいて、 内側のループを抜けたときに 1回も交換していなかったら、もう比較する必要も無い、と判定してやるのはどうでしょう?

Swap カウンタの導入 (製品コード)

- Swap() メソッドを改修します。
- こんどのカウンタは、 さっきのプローブ用のとは違って、 どんな呼ばれ方をするか分からない製品コードの部分になりますから、 static ではダメです。 ちゃんとメンバ変数にしておきましょう。

```
private int _swapCount;

private void Swap(ref int n, ref int m) {
        this._swapCount++;

        int work = n;
        n = m;
        m = work;
}
```

- そして、内側のループに入る前に _swapCount をゼロクリアして、出てきたときにゼロのままだったら、ソート終了と判定します。

```
public int[] Sort(int[] input) {
    if (input.Length == 1) return input;

    for (int stop = input.Length - 1; stop > 0; stop--) {
        this._swapCount = 0;
        for (int i = 0; i < stop; i++) {
            if (IsNotInOrder(input[i], input[i + 1])) {
                Swap(ref input[i], ref input[i + 1]);
            }
        }
        if ( this._swapCount == 0 )    break;
    }
    return input;
}                                                    // -------- src04.sln --------
```

宿題 (その3)

- まだ不足しているテストケースがある。
- 降順にもソートせよ。
  昇順/降順の切り替えをどうやるかも決定せよ。 (ただし、 既存のテストコードは、そのまま通ること。)
- int 以外の型も使えるように拡張せよ。 (数値だけでなく、 文字列はどうか?)
- ソートのロジックを違うものに切り替えられるようにせよ。 (デザインパターンの練習)

補足: 例外が出るところをテスト

- 製品コードから例外が出る (そういう仕様である) ことを確認するためのテスト
- ExpectedException 属性を使うか、 あるいは、 次のように try ～ catchする。

```
WankumaSorter sorter = new WankumaSorter();
try {
        int[] result = sorter.Sort(null);
        Assert.Fail("この行に来たらアウト!");
}
catch (ArgumentNullException) {
        // OK! --- ここで例外メッセージの検査なども出来る
}
```

参考

- URL
  - **NUnit入門 Test Firstのススメ** (川俣 晶)
    http://www.atmarkit.co.jp/fdotnet/tools/nunit2/nunit2_01.html
  - **「テスト駆動開発」はプログラマのストレスを軽減するか?** (川俣 晶)
    http://www.atmarkit.co.jp/fdotnet/special/tdd/tdd_01.html
  - **.NET開発者のためのリファクタリング入門** (川俣 晶)
    http://www.atmarkit.co.jp/fdotnet/special/refactoring/refactoring_01.html
- 書籍
  - **テスト駆動開発入門** (ケント ベック)
  - **Microsoft.NETでのテスト駆動開発** (ジェームス・ニューカーク)
  - **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)