世界はオブジェクトの海に浮かぶ

.NET Frameworkで楽しむオブジェクト指向

第8回 テスト駆動開発のススメ

επιστημη
えぴすてーめー

しばしおさらい

前回はデバッグの手法に関するトピックとして、System.Diagnostics名前空間に定義されたDebugおよびTraceの使い方を紹介しました。両者のWriteメソッドで処理の進行状況を逐一レポートさせることができますし、Assertメソッドをうまく使えば「何かヘンなことが起こった」ことを直ちに知ることができます。デバッグで肝要なのは原因に極力近いところで現象を捕まえることです。原因から遠く離れたところで現象を捕まえると、そこから原因を絞り込むのに余計な労力を費やすことになります。

加えてUnit Test Frameworkのひとつ、NUnitを使った単体テスト、そしてTDD（Test Driven Development：テスト駆動開発）についてそのあらましを述べました。全部書いてからテストしバグを潰すのではなく、先にテストを書き（これが書けないようなら設計が甘い）、そのテストをパスするように実装することで、確実に動くコードを書こうというアプローチです。場合によっては作りたいコードそのものよりテストコードのほうがデカくなっていささかうんざり、かもしれません。けれど一旦テストを書いてしまえば以後は常にテストに守られた状態を維持しながら変更／拡張できるわけで、決して無駄にはならないはずです。ちゃんと動くアプリケーションを作りたいならテストは不可欠であって、ならば書き上げてからテストするより、もっと早期にテストを用意して設計／実装にも活用しちゃえってわけね。

Level

1 2 3 4 5

Technology Tools

- ◯ Visual Basic
- ✓ Visual C#
- ◯ Visual C++
- ◯ SQL Server
- ◯ Oracle
- ◯ Access
- ◯ ASP.NET
- ✓ Other：
  ↓
  NUnit 2.2

Samples

この記事で取り上げたソースコードおよびサンプルプログラムは、
http://www.shoeisha.com/mag/windev/
からダウンロード可能です。

実装とインターフェイスの微妙なカンケイ

アプリケーションの構築は建築や機械と似たところがあります。原料から部材を作り、それを組み合わせて部品を作り、部品を組み合わせてモジュールを、モジュールを組み合わせて……ってな具合に小さな／基本的なものを組み上げて次第に大きなものへと階層化されていきます。最終的には各部品がすべて揃い、収ま

るべきところに収まってないと正常に機能することはありませんが、ものによっては部品がなくてもそれより上の階層を組み上げることができます。つまり事前にきちんと設計しインターフェイスが決まっていればいい。

正しく描かれた設計図によって部品に開けられたネジ穴の位置が決まっているのなら、その部品を組み込むモジュールは（その部品がまだ完成していなくても）ネジ穴を開けて製造工程を進めることができます。あるいは部品、たとえばモーターがまだ完成していなくても、外形（インターフェイス）だけは寸分違わぬハリボテ（鉄の塊）を使って組み上げることができます。ただし当然のことながらスイッチ入れてもウンともスンとも言いませんけどね。

これとまったく同様のことをプログラミングでもできますし、実際にやっているわけです。

たとえばあなたが掛け算クラス「Multi」の実装担当だとしましょうか。Multiには実行メソッドが定義されますよね。

```
class Multi {
   public int eval(int x, int y) {
      // x * yを返す
   }
}
```

2数の積を求めるメソッドevalの実装にあたって、「x * y」はxをy回加えればいい、足し算クラスがあれば、それを複数回呼び出すことで掛け算が実現できると考えたあなたは同僚に足し算クラス「Add」を作ってくれるように頼みました。同僚は快く引き受けてくれましたが「今すぐに」は無理とのこと。その代わりインターフェイスだけは決めてくれました。机の上に彼のメモが残されています。

```
/* これ使って仕事進めておくれ! */
public interface IOperation {
   int eval(int x, int y);
}

public class Add : IOperation {
   …今忙しいからゴメン…
}
```

だそうです。彼は今の仕事に手一杯で、あなたが頼んだAddが出来上がるのはしばらく後になりそう。だからといってリリースしてくれるまで遊んで待ってるわけにはいきません。Addを受け取り次第、ガチンと繋いでリリースしないとスケジュールを狂わせてしまいます。あなたが作るMultiを心待ちにしているプログラマもいるのだし。

そこでIOperationを実装し、あたかも本物のAddであるかのようにふるまう“似非Add”を作り、それを使ってあなたの担当であるMultiの実装とテストを進めることにしましょう。

Multiの実装はこんな感じになりますか。

```
public class Multi {

   private IOperation operation;

   public void SetOperation(IOperation op) {
      operation = op;
   }

   // x * yを求める
   public int eval(int x, int y) {
      // 必要ならば符号反転
      if ( y < 0 ) { y = -y; x = -x; }

      // result←result + x をy回繰り返す
      int result = 0;
      for ( int i = 0; i < y; ++i ) {
         result = operation.eval(result,x);
      }
      return result;
   }
}
```

似非メソッドを作るコツ

さて、このMultiがちゃんと動いてくれるかをテストすべく、IOperationを実行した似非Add「MockAdd」を作らなくては（mockはニセモノ／真似事の意）。このMockを作るにあたって大事なことは、あくまで“見かけ上本物のようにふるまう”ことであり、中身を本物そっくりに実装してはいけません。この例では単に二数を加算するだけですが、実際にはもっと複雑な処理を行な