第 2 回 プログラミング基礎ゼミ

ゼミ担当者　：　山本 啓二，狩野 浩一，下村 大輔
指導院生　　：　實田 健，澤田 淳二，金 美和
開催日　　：　2003 年 6 月 5 日

ゼミ内容: 本ゼミではテストプログラミング，その中でも特にテストファーストと呼ばれる手法について説明する．テストファーストとは，まず，モジュールがどのような役割を持つべきかを考え，そのモジュールに対するテストを作成した後，実際のモジュールを作成し，テストを行い，モジュールが正常に動作しているかを確認するものである．まず，テストを行うことの重要性について説明した後，テストを支援するためのフレームワークを説明する．その後，簡単なモジュールの作成を通して，テストファーストについての理解を深める．

1　はじめに

プログラムを作成する際に，バグは付き物である．あるプログラムを作成し，そのプログラムが正常に動作しなかったという場合，デバッグが行われる．その際に，作成したプログラムの各部品 (**モジュール**) がどのような役割を持つべきかを明確にしておかないと，デバッグの際に効率的にバグの原因を見つけだすことができず，デバッグに長い時間がかかってしまう．最悪の場合，ソースコードを一行一行トレースし，プログラムが正常に動作しているかの確認作業をしなければいけなくなる．また，一見正常に動作しているように見えても，実は自分が期待しているような動作が行われていないということもあり得る．この場合，バグを発見するのはより困難となる．

バグを発見するためには，モジュールを作成したら，そのモジュールに対してテストを行うことが有効である．その場合，モジュールに対してどのようなテストを行えばよいかが問題となる．また，テストを行うのは面倒な作業であるので，多くのプログラマはプログラムを走らせてみて,「ちゃんと動いている (ように見える) のだから，テストはしなくていいや」と考えてしまいがちである．

そこで，モジュールを作成してからそのモジュールに対するテストを作るのではなく，まず，モジュールが持つべき機能を考え，それに対するテストを作ってしまってからモジュールを作成するという手順にすることで，各モジュールが持つべき機能を明確にし，モジュールのテストも効率的に行うことが可能である．このような手法をテストファーストと呼ぶ.

2　テストファーストによるテストプログラミング

2.1　テストファースト

XP(eXtreme Programming) において，**テストファースト**と呼ばれる手法が提唱された．これは，モジュールのコーディングに入る前に, そのモジュールに対するテストコードを先に書く手法である．テストファーストには，次のようなメリットがある．

- 仕様の明確化

  モジュールを書く前にテストが作れないということは，そのモジュールの振る舞いがはっきり理解されていないということである．逆の言い方をすると，テストファーストならば，モジュールを書くときには振る舞いがはっきりしているので，モジュールを書くことに専念できる．

- すべてのモジュールに対してテストが作成できる

  テストファーストならば，すべてのモジュールに対するテストが必ず存在することになる.「動いたからテストは書かなくていいや」という考えを防ぐためにもテストファーストは有効である．

- ドキュメントとしての役割

  テストは最高のドキュメントである．テストは，モジュールの包括的な振る舞いを確認しているので，テストを見ることでモジュールの使用法，仕様が理解できる．

- シンプルな設計

  テストを先に書くことで，シンプルな設計／実装を行うようになる．

- ゴールの明確化

  テストを先に書くことで，コーディングのゴールを知ることができる．ゴールとはテストに通ることである．これはプログラマにとって精神衛生上良い効果をもたらす．

テストファーストでは，次のステップでモジュールを作成することになる．

1. モジュールがどのような機能を満たすべきかを考える
2. モジュールの機能に対するテストを考え，実装する
3. モジュールを実装する
4. テストを行う
5. テストが成功しなかったら，デバッグを行う．その後，再びテストを行い，テストを通過するまで繰り返す

2.2 テスティングフレームワーク

テストを行う際に，テストを自動化することは重要である．自動化されたテストならば，テストを「いつでも」「素早く」「すべて」実行することができる．printf() や System.out.println() を用いた目視による確認やデバッガを用いたステップ実行によるチェックは自動化が行われていないので，テストには不十分である．

そこで，テストを支援するために**テスティングフレームワーク**が提案されている．テスティングフレームワークを用いることには次のようなメリットが存在する．

- テストが簡単に実装できる

  テストは簡単に，なるべくプログラマにかかる負担が少ない形で実装するのが望ましい．テスティングフレームワークを利用することで，最小限のコーディングでテストを構築できる．

- テストが統一的に作成できる

  テスティングフレームワークを用いない場合，テストは各プログラマがそれぞれのやり方で実装してしまう．フレームワークという枠組みがあることで，統一的かつ一貫した方法でテストを行うことができる．

- テストコードとモジュールコードの分離

  テスティングフレームワークでは，例えば，モジュール「Hoge」に対して、テスト「HogeTest」を作成する．このことにより，テストコードとモジュールコードを分離することができる．

2.3 テスティングフレームワークによるテストの作成・実行方法

様々なプログラミング言語について，次のようなテストを行うためのテスティングフレームワークが提供されている．

- C 言語：Check(http://check.sourceforge.net/)
- C++：CppUnit(http://cppunit.sourceforge.net)
- Java：JUnit(http://www.junit.org/)
- Perl：Test::Unit
- Ruby：Test::Unit

これらの各フレームワークは，JUnit をベースとしており，基本的な利用方法は同じである．JUnit とは，Java のクラスが正しく動作することをテストするためのオープンソースのテスティングフレームワークである．

ここでは，JUnit を例に，テストを作成し，実行を行うための手順について説明する．

1. テストを行うためのクラスの作成

   JUnit では，TestCase と呼ばれるクラスが用意されている．このクラスを継承することで，テストのためのクラスを作成する．

```
import junit.framework.*;

public class HogeTest extends TestCase {
}
```

2. テストの前準備と後始末

   複数のテストで必要となる変数がある場合は，前準備として setUp() メソッドを，後始末として tearDown() メソッドをオーバーライドする．前準備，後始末の必要がない場合はオーバーライドをする必要はない．

   例として,「これからテストを行いますよ」ということを出力する setUp() メソッドを記述する．

```
protected void setUp() {
    System.out.println("Now, I start test for Hoge class");
}
```

3. モジュールに対するテストの作成

   テストを行うためのメソッドは，testAdd() のように,「test テスト名 ()」とする．この際，1 つのメソッドに全てのテストを記述するのではなく，チェックしたい項目ごとにテストを作成するようにするとよい．

   テストの基本は，ある入力に対して，自分が期待している出力が得られているかをチェックすることである．つまり，足し算を行うメソッドがあり，“add(1, 2)” という入力に対して，3 という出力を期待するのならば，テストは次のようになる．

```
public void testAdd() {
    assertTrue(add(1, 2) == 3);
}
```

   条件式が成り立つかどうかのテスト以外にも，出力が期待される出力と等しいかを確認するテストも存在する．その場合は，次のようなテストコードになる．

```
public void testAdd() {
    assertEquals(new Integer(3), new Integer(add(1, 2)));
}
```

   独自のクラスについて，assertEquals() によるテストを用いたい場合は，そのクラスについて，equals() メソッドと toString() メソッドをオーバーライドする必要がある．そうしないと，assertEquals() メソッドは，期待される出力と実際の出力が等しいかどうかの確認が行えないため，テストは失敗してしまう．

4. テストスイートの作成

   一連のテストをまとめたものを**テストスイート**と呼ぶ．JUnit では，作成したテストのためのクラスに対して,

```
TestSuite suite = new TestSuite(HogeTest.class);
```

   とすることで，容易にテストスイートを作成することが可能である．

5. テストの実行

   作成したテストスイートを実行するためのものとして，TestRunner と呼ばれるクラスが用意されている．このクラスを用いることで，テストを実行することが可能である．テストを実行するためのコードは，次のようになる．

```
TestSuite suite = new TestSuite(HogeTest.class);
junit.textui.TestRunner.run(suite);
```

テストを実行するためのユーザインターフェースは，コンソールへの出力以外にも，AWT や Swing による GUI を用いることも可能である．

3 簡単なテストプログラミングの実例

それでは，具体的な例題を用いて，テストプログラミングを実践する．ここでは，Java で複素数を扱うクラスを作成することを考える．

なお，テストクラスのコンパイル，テストの実行を行う際には，配布されている JUnit のパッケージを展開して得られる junit.jar に CLASSPATH が通っている必要がある．`~/local/junit3.8.1` に junit.jar があるという場合は，次のように CLASSPATH を指定する．

```
export CLASSPATH=~/local/junit3.8.1/junit.jar:.
```

3.1 モジュールが持つべき機能

簡単のために，次のような機能だけを持つものとする．

- 実数部と虚数部がある
- 複素数の加算と減算ができる

3.2 モジュールに対するテスト

3.1 節で考えたモジュールに対するテストを考える．このモジュールには，次のようなテストが必要である．

- 実数部と虚数部を返すメソッドがちゃんとできてるか
- 足し算と引き算がちゃんとできてるか

上記の各項目に対するテストを実装する．テストコードは，Fig. 1 のようになる．

```
import junit.framework.*;

public class ComplexTest extends TestCase {
    private Complex c1, c2;

    protected void setUp() {
        c1 = new Complex(10.0, 3.0);
        c2 = new Complex(22.0, 44.0);
    }
    protected void tearDown() {
        //nothing to do
    }
    public void testComplexCreate() {
        assertTrue(c1.re() == 10.0);
        assertEquals(new Double(3.0), new Double(c1.im()));
    }
    public void testComplexOperator() {
        Complex ca = new Complex(32.0, 47.0);
        Complex cs = new Complex(-12.0, -41.0);
        assertEquals(ca, c1.add(c2));
        assertEquals(cs, c1.sub(c2));
    }
    public static void main(String[] args) {
        TestSuite suite = new TestSuite(ComplexTest.class);
        junit.textui.TestRunner.run(suite);
    }
}
```

Fig. 1　複素数クラスに対するテストコード

3.3 モジュールの実装

複素数クラスの実装を行う．複素数クラスの実装コードは，Fig. 2 のようになる．

```
public class Complex {
    private double re, im;

    public Complex(double r, double i) {
        re = r;
        im = i;
    }
    public double re() {
        return re;
    }
    public double im() {
        return im;
    }
    public Complex add(Complex a) {
        return new Complex(re + a.re(), im + a.im());
    }
    public Complex sub(Complex a) {
        return new Complex(re + a.re(), im + a.im());
    }
    public String toString() {
        return re + "+" + im + "i";
    }
    public boolean equals(Object o) {
        Complex c = (Complex)o;
        return re == c.re() && im == c.im();
    }
}
```

Fig. 2　複素数クラスの実装コード

3.4 テストの実行

テストの実装，モジュールの実装が完了したので，テストを実行してみる．テストの結果は Fig. 3 のようになる．

```
$ javac Complex.java ComplexTest.java
$ java ComplexTest
..F
Time: 0.018
There was 1 failure:
1) testComplexOperator(ComplexTest)junit.framework.AssertionFailedError: expecte
d:<-12.0+-41.0i> but was:<32.0+47.0i>
        at ComplexTest.testComplexOperator(ComplexTest.java:21)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcces
sorImpl.java:25)

        at ComplexTest.main(ComplexTest.java:25)

FAILURES!!!
Tests run: 2,  Failures: 1,  Errors: 0
```

Fig. 3　テスト結果 (失敗)

FAILURE!!!と出力されているので，テストに失敗したことになる．失敗箇所は，ComplexTest.java の 21 行目で，

-12.0+-41.0i が期待されているのに，実際の出力は，32.0+47.0i であるとなっている．Fig. 1 の 21 行目を見ると，

```
assertEquals(cs, c1.sub(c2));
```

とあるので，sub() メソッドが正しく動作していないことがわかる．

3.5 実装コードの見直し

テストに失敗したので，実装コードの見直しを行う．Fig. 2 を見ると，sub() メソッドにおいて，

```
public Complex sub(Complex a) {
    return new Complex(re + a.re(), im + a.im());
}
```

となっている部分に気づく．どうやら，add() メソッドのコードをコピー&ペーストした後，+を-に修正し忘れたようだ．コードの修正を行い，もう一度テストを行う．テスト結果は Fig. 4 のようになる．

```
$ javac Complex.java
$ java ComplexTest
..
Time: 0.013

OK (2 tests)
```

Fig. 4　テスト結果 (成功)

今度は無事，全てのテストを通過した．これで，複素数クラスは完成である．