第 1 回 テストゼミ

ゼミ担当者　：　川崎 考蔵，宮村 幸祐，谷口 真一
指導院生　：　平尾 洋樹，千田 智治
開催日　：　2007 年 5 月 25 日

ゼミ内容: 本ゼミでは，プログラミングにおいて必要とされる，テストの概念と JUnit について学ぶ.

1　はじめに

本ゼミでは，システム開発におけるテスト工程についての説明を行う. テストの必要性，またその目的は何であるかを明らかにする．システム開発におけるテストには，単体テスト，結合テスト，統合テスト及び機能テストといったものがあるが，本ゼミでは単体テストに焦点をあてて説明を行う．そして，Eclipse に備わっているテスト実行環境である JUnit の使い方についての説明を行う.

2　テストとは

システム開発において，プログラムを実装するだけではなく記述したソースコードにバグがないかどうかを検討し，システムの品質を確保する必要がある．そしてシステムの品質を保つためにテストという工程を行う．テストの目的は以下の通りである.

- バグを発見する
  一つずつバグを発見し，一つずつ修正することで，システムの品質は確保される.
- 工程が完了したことを確認する
  テスト仕様書に基づいてテストを行うことで，システム開発の一つ一つの工程が完了したことを，誰でもわかるような明瞭な形で示す.
- システムをより優れたものに改良する
  テストによりシステムの現在の状態を把握して，改良を加える.

テスト工程の中で一番重要なものが単体テストである．その理由としては，組み込みソフトは大規模化，複雑化しており，結合テストの工程でバグのほとんどを検証することは不可能になっている為，単体テストの持つ，開発の早い段階で潜在的なバグが発見できること，小さなプログラムの各段階で動作確認を行うことにより結合テストの工数を削減することが出来るというメリットは非常に効果的であるためである．また，単体テストの他のメリットは以下の通りである.

- テストの結果がソフトの品質をを示す「客観的，定量的」なデータとなる.
  入出力テストや網羅率などの定量的で客観的のある検証データとなり，ソフト品質に対する指標が明確になる.
- 関数設計者自身による「網羅率」を意識したテストにより，ソフト品質を向上することが出来る.
  開発者自身が自分の開発した関数を深く見直すことが出来，気が付かなかったバグを発見することが出来る.

現在行われている手法として主に挙げられるものはテストファーストという手法である．これは，仕様に合致する単体テストを先に書き、その単体テストに通るようなコードを書くという手法である．この手法の特徴は以下の通りである.

- 先にテストの設計を行うので，ソースコードの満たすべき機能やプログラムの設計仕様を示すことが出来る.
- 容易にテストを行うことが出来る為，テストが何回も行え，開発効率は上がる.

3　JUnit

Eclipse には JUnit というテスト実行環境が備わっている．JUnit は，1997 年に Eric Gamma と Kent Beck が開発した Java プログラム単体テスト用のフレームワークのことで，Java のプログラム単位であるクラス毎に単体テストを行うものである．また，作成したテストケースとプログラムの実行結果が合っているかをチェックする機能を提供する．JUnit によって共通のテストフレームワークをもつことで，他人のテストプログラムの修正を容易にする．現在プログラムに対する要求が複雑化しており，人の手によるテストの精度には限界がある為，テストプログラムを作成し，自動化テストを行うことが望ましいといえる．JUnit は，その問題に対処しており，テスト仕様が作成できればテストを自動化させることが可能となっている．また，JUnit では，複数のテスト

ケースをまとめて実行することが可能である．更に，テスト仕様があれば，プログラミング完成のゴールとすることが出来る．JUnit には他に次のメリットがある．

- テストが簡単に実装できる
  JUnit はテスト用のクラスを継承してテストクラスを作成しているので，最小限のコーディングで単体テストを構築できる．

- テストが統一的に作成できる
  フレームワークという枠組みがあることで，統一された方法でテストを行うことができる．

- テストと実装コードの分離
  JUnit では実装クラス「Sample」に対して，テストクラス「SampleTest」を作成する．このことにより，テストコードで実装コードを分離することができる．

- リグレッションテストが容易に行える
  リグレッションテストとは，プログラムのバグを修正したことによって，そのバグが取り除かれた代わりに新しいバグが発生していないかどうかのテストのことである．テストの自動化により，リグレッションテストが容易に行えるようになる．

4 JUnit の実行手順

実際に JUnit を動かしてみる．今回は簡単な足し算を行う AddNum クラスのテストを作成する．

```
public class AddNum {
    public int addNum(int a,int b){
        int answer = a+b;
        return answer;
    }
}
```

4.1 JUnit のテストクラス作成

Eclipse を使用して，JUnit のテストクラスを作成する．作成方法は以下の手順となる．

1. Java プロジェクトを作成する．
2. 作成したプロジェクトに新しいクラス「AddNum」を作成する．このとき，「public static void(String[] args)」と「継承された抽象メソッド」にチェックを入れる．(Fig. 1 参照)
3. パッケージエクスプローラーでクラス「Addnum」を右クリックし，「新規」→「JUnit Test Case」を選択する．(Fig. 2 参照)

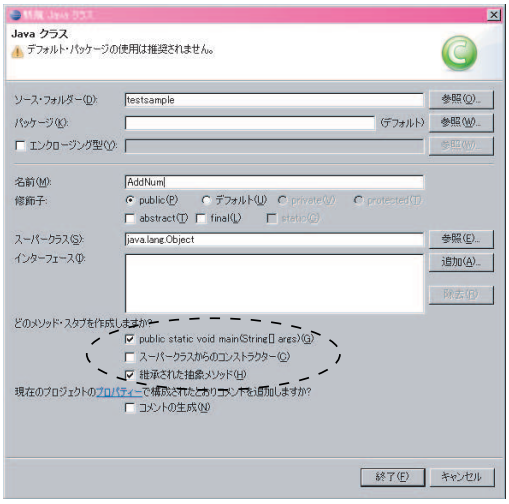

Fig. 1　クラスの作成（出典：自作）

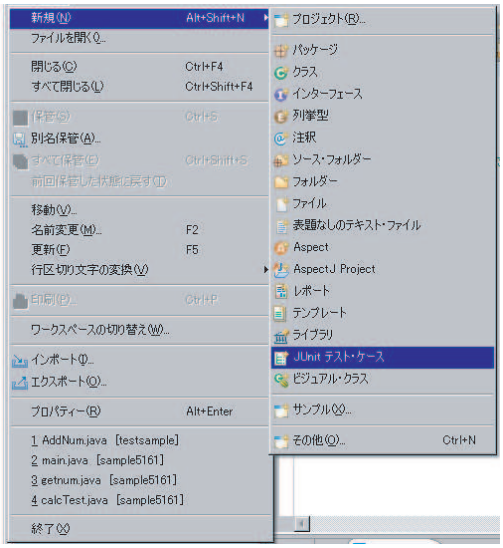

Fig. 2　JUnit Test Case の作成（出典：自作）

4. 新規テスト・ケース画面が出てくるので「はい」をクリックする．(Fig. 3 参照) すると,JUnit ライブラリ junit.jar のパスがビルドされる．(Fig. 4 参照)

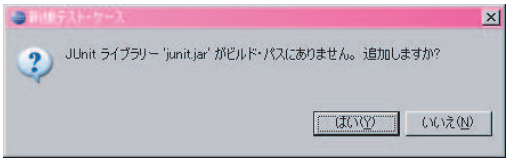

Fig. 3　新規テスト・ケース画面（出典：自作）

5.　新規 JUnit テスト・ケース画面が出てくるので「終了」をクリックする．(Fig. 5 参照)

以上の処理で，テストクラス「AddNumTest」が作成される．

今回は以下のようにコードを書く．

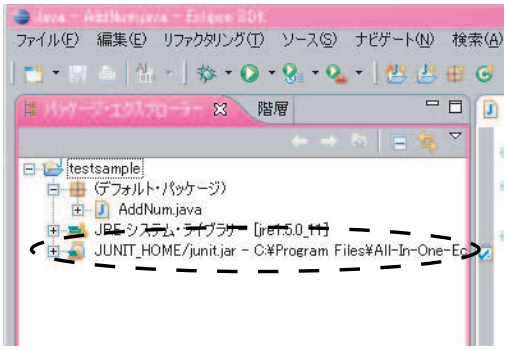

Fig. 4　JUnit.jar へのパス（出典：自作）

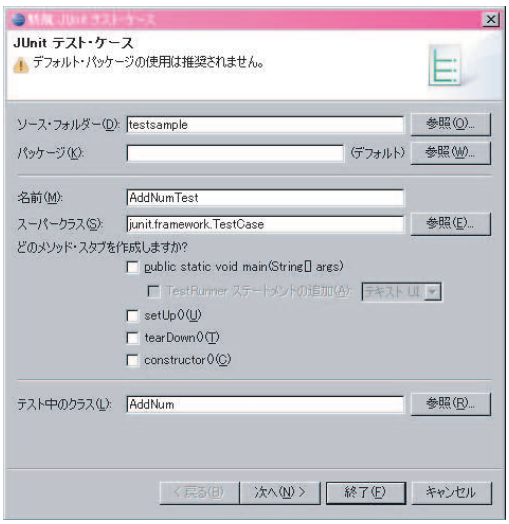

Fig. 5　テストクラスの作成（出典：自作）

```
import junit.framework.TestCase;

public class AddNumTest extends TestCase {
    public void testAddNum(){
        int tn1 = 17;
        AddNum check = new AddNum();
        int test = check.addNum(8,9);
        assertEquals("計算ミス",tn1,test);
    }
}
```

実際にテストクラスを作成するときは，テストケースのメソッドは必ず“**test**”からはじめなければならない．今回は“**testAddNum()**”という風になる．

4.2　JUnit の実行方法

次に，テストを実行する．実行方法は以下の手順となる．

1. パッケージ・エクスプローラーのテストクラス「AddNumTest」を右クリックし,「実行」→「JUnit テスト」を選択する．(Fig. 6 参照)

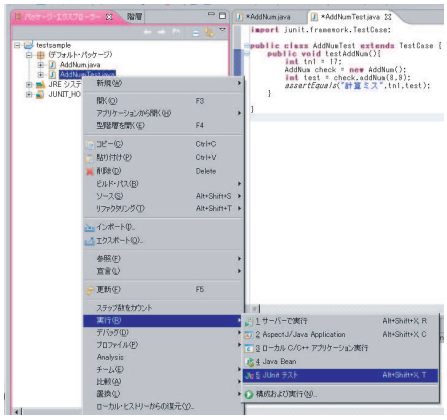

Fig. 6　JUnit の実行（出典：自作）

2. JUnit ビューが起動し，JUnit の結果が Fig. 7 のように表示される．

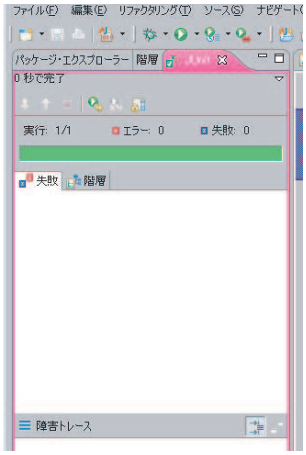

Fig. 7　JUnit の実行結果（成功）（出典：自作）

緑色のバーが表示されればそのテストは成功したことになる．一方，赤色のバーが表示されればそのテストは失敗したことになる．この例ではテストは成功である．

試しにテストクラス「AddNumTest」を以下のように変更する．

```
import junit.framework.TestCase;

public class AddNumTest extends TestCase {
    public void testAddNum(){
        int tn1 = 17;
        AddNum check = new AddNum();
        int test = check.addNum(10,9);
        assertEquals("計算ミス",tn1,test);
    }
}
```

そうすると Fig. 8 のように表示され，テストが失敗したことが分かる．

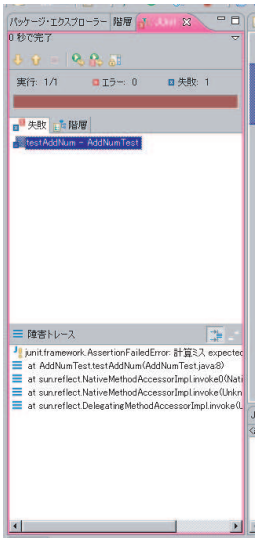

Fig. 8　JUnit の実行結果（失敗）（出典：自作）

このとき，下部に以下のようなメッセージが出てくる．

```
junit.framework.AssertionFailedError:
計算ミス expected:<17> but was:<19>
```

これは期待値 17 に対し，結果が 19 であったため，検証が失敗したことを意味する．

5　JUnit のメソッド

5.1　アサーションメソッド

前章の AddNumTest クラスでは assertEquals メソッドを利用してテストを行った．assertEquals メソッドは 2 つの値が同じかどうかを判断し，違う値ならメッセージを返すという機能を持っている．JUnit で用意されているアサーションメソッドには Table 1 のようなものがある．

5.2　独自のメソッド

JUnit には，アサーションメソッドの他に独自に定義されるメソッドがある．その中で代表的なのが setUp メソッドと tearDown メソッドである．

- setUp()
  テストの実行前に呼び出されるメソッドである．複数のテストケースがあるときは，各テストが実行される前に実行されることになる．そのため，テスト実行前にテスト環境を設定するのに用いられる．

- tearDown()
  テストの実行後に呼び出されるメソッドである．setUp と同じように，各テストが実行された後に実行される．

setUp と tearDown は Fig. 9 のように実行される．setUp と tearDown は testXXX と testYYY と testZZZ が実行されるたびに呼び出されることになる．

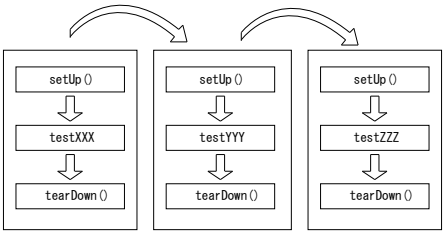

Fig. 9　setUp と tearDown の使い方（出典：自作）

例えばデータベースにアクセスしてテストを行う場合，setUp でデータベースへの接続，tearDown でデータベースへの接続の切断を行えば，コードが短くすることが可能となる．

Table 1　アサーションメソッド（参考文献 [1] より参照）

メソッド	判定	メッセージ
assertEquals(arg1, arg2)	arg1，arg2 が同じ値	なし
assertEquals(message, arg1, arg2)	arg1，arg2 が同じ値	あり
assertTrue(boolean arg1)	arg1 が true	なし
assertTrue(message, boolean arg1)	arg1 が true	あり
assertFalse(boolean arg1)	arg1 が false	なし
assertFalse(message, boolean arg1)	arg1 が false	あり
assertNull(Object arg1)	arg1 が null	なし
assertNull(message, Object arg1)	arg1 が null	あり
assertNotNull(Object arg1)	arg1 が null ではない	なし
assertNotNull(message, Object arg1)	arg1 が null ではない	あり
assertSame(Object arg1, Object arg2)	arg1 = arg2 が true	なし
assertSame(message, Object arg1, Object arg2)	arg1 = arg2 が true	あり
assertNotSame(Object arg1, Object arg2)	arg1 = arg2 が false	なし
assertNotSame(message, Object arg1, Object arg2)	arg1 = arg2 が false	あり
fail()	テスト強制失敗	なし
fail(message)	テスト強制失敗	あり

参考文献

1) 快適な XP ドライビングのすすめ　第 5 回
   http://www.atmarkit.co.jp/im/carc/serial/xpd05/xpd05.html

2) JUnit-TECHSCORE-
   http://www.techscore.com/tech/Others/JUnit/index.html

3) Java の道：Eclipse(11.JUnit の利用)
   http://www.javaroad.jp/opensource/js_eclipse9.htm