JUnit 入門

NCS　XP 研究会

日本コンピューター・システム株式会社

(http://www.ncs.co.jp/)

はじめに

Junit とはについて、ここでは最上段に構えて話をするわけではありません。どちらかと言えば、初心者のために Junit とはどのようなもので、どう使うのかという観点で説明しています。

まずは、少し横道にそれたようなお話から入りましょう。

JUnit の文字を見ると J と Unit に分かれています。J は誰でも簡単にわかるうに Java ですね。では、Unit はというと、ユニットを組むとかオプションユニットとか何かまとまりの単位を表わしているときに良く使いますね。確かに、辞書で引けば Unit は単位という言葉で説明されています。

そうですね、昔、汎用機の時代には、ユニット・テストと読んでいましたね。ユニット・テストは、サブルーチンやプログラムをそれぞれ単体もしくは数個をひとつとしてテストすることをユニット・テストと呼んでいました。では、その頃、ユニット・テストをどうしていたのでしょうか。それ自身が単独で実行できるものは、入力側のデータを予め用意しておいて、実行し、その結果を印刷してチェックしました。では、そうでないものはどうしたのかというと、実は、テストプログラムというものを作成し、テストすべきプログラムを結合してテストをしたのです。汎用機の世界においても、テストにおけるサポートツールが提供され、最終的にはそのツールを用いてテストするようになりました。

さて、横道にそれるのはこの辺にしておいて、本題のとにもどりましょう。この本では、JUnit について以下の項目でお話します。

1. JUnit とは
2. JUnit の特徴
3. JUnit のインストール方法
4. Junit の使い方

たぶん、これらの項目は皆様がもっとも知りたいものでしょう。では、何故このようなもりが書けるのでしょうか。それは、JUnit について日本 XP ユーザグループで出版することになり、「JUnit とは」の原稿の一部を私達がお手伝いできることになったからです。

最後に、この機会を与えてくださった日本 XP ユーザグループに感謝するとともに、読者の皆様にとって良き入門書となることを祈ります。

（NCS　XP 研究会一同）

目　　　　　　次

はじめに I

1. JUNIT 1

1.1. JUNIT とは 1
1.2. JUNIT の仲間には何がある 1
1.3. JUNIT は何故必要か 1

2. JUNIT の特徴 3

2.1. JUNIT を使えば、テストにかかる時間を短縮できます 3
2.2. JUNIT を使えば、テストを簡単に書くことができます 3
2.3. JUNIT を使えば、品質の向上ができます 4

3. JUNIT インストール 5

4. JUNIT の使い方 7

4.1. まず、使ってみよう 7
4.2. テストの拡張 8
4.3. TESTRUNNER によるテストの自動実行と結果表示 11
例 1 11
例 2 11
テストの起動 11

あとがき 14

1. JUnit

1.1. JUnit とは

JUnit とは、ソフトウェアのテストを簡単にかつ体系的に行なえるようにするために作成されたテスティングフレームワークの Java 版です。ソフトウェアのテストは、開発者の視点でプログラムの動作をテストする Unit Testing とユーザ側の視点で機能面のテストをする Functional Testing に分けられますが、テスティングフレームワークは Unit Testing をカバーしています。

このテスティングフレームワークでは、開発対象言語と同じ言語でテストコードを記述することから、各言語に対応したテスティングフレームワークがあり、Java 用として JUnit（Java Unit）が 1997 年に Kent Beck と Eric Gamma により開発されています。現在は、www.junit.org が公式サイトになっており、オープンソースソフトウエアとしてリリースされています。最新版は、2001 年 5 月 27 日現在で JUnit3.7 です。

1.2. JUnit の仲間には何がある

他言語用のテスティングフレームワークとしては、VB用のVBUnit、C++用のCppUnit、Ruby用のRubyUnit、Delphi用のDelphiUnit等があります。また、IDE向けには、JUnit¥ With¥ JBuilderやJUnit¥ With¥ Visual¥ Ageがあり、JUnitの拡張版として、EJB向けにEjbUnitTest、JUnitEE等があります。また、これらと組み合わせてテストを行なうWeb系テスト用としてHTTPUnit、Servlet用にServlerUnitTest、JakartaAntから実行するJavaUnitAndAnt等があります。

1.3. JUnit は何故必要か

eXtream Programming においては、コードを書く前にテストを書くこと、テストは各単体(Unit)プログラム(Javaでは、Classもしくはその集合)毎にこまめに行なうこと、という方法が重要なポイントとなっています。小さなプログラムを集積して大きなプログラムが作られて行く中では、各小さなプログラムの段階できっちり動作を確認することが、結合テスト工数を削減でき、品質の高いソフトウェアを高い生産性で作れる、という考え方です。

また、新しいプログラムが出来たら、そのプログラムだけをテストするのではなく、既にテスト済みのプログラム群に加えて、テスト済みプログラムへの影響は無いか、ということを確認することで、どのプログラムに問題があるのかを早期発見できます。

さらに、XPにおいて注目すべきプラクティスであるリファクタリングでは、テスト済みのプログラムでもコードを洗練させるために、手を入れて行きますので、それによって正常に動作していたプログラムが手を入れたことによって動作しなくなったかもしれない、ということも即時に確認(テスト)する必要があります。

このように、XPにおいては、テストというものの位置づけは非常に重要かつ、日常(一日に何回も)実行するものです。これらのプラクティスを実践するためには、プログラマにとって、これをより効率的に、テストを簡単に実行できることがより重要と言えます。

テストは従来、まずコードを書き上げ、それをテストするためのプログラム(テストドライバ)を書き上げて、そのテストドライバを利用してテストを行ないました。

ソフトウェアのテストを効率良く行なう上でテストドライバをその都度作成することは効率的ではなく、面倒な作業です。
それらをより簡易に取り扱うことを目的として、テスティングフレームワークが作成され、Javaに対応するJUnitが作成されたのです。テスティングフレームワークでは、テストケースの構成を決める方法が決まっており、テストを自動的にまとめて実行する機能や結果を即時に表示する機能が用意されていますので、コードを変更してすぐ自動実行し、他のコードへの影響が無いかどうかを即時に確認することができます。JUnitはリファクタリングのための道具とも言えるのです。

2. JUnitの特徴

JUnitの特徴を以下の３つの視点からお話しましょう。

- JUnitを使えば、テストにかかる時間を短縮できます
- JUnitを使えば、テストを簡単に書くことができます
- JUnitを使えば、品質の向上ができます

2.1. JUnitを使えば、テストにかかる時間を短縮できます

JUnitはテストを自動で実行でき、テスト結果も自動的にチェックしてくれます。結果が正しいか間違っているか、即座にフィードバックされます。

また、視覚的に分かりやすく表示してくれます。テストを通れば緑色、結果が間違っていたら赤色のプログレスバーが表示されます。また、どこが間違っているのかを教えてくれるので、もうソースコードを1行1行追っかけて、間違っている箇所を探し出すということをしなくてよいのです。

JUnitを使うことでこれまで行ってきた、テスト結果と正しい答えを手作業で確認するという手間を省略できます。

2.2. JUnitを使えば、テストを簡単に書くことができます

「テストを書く」ということが、非常に面倒であったり大変時間がかかったりするとしたら、どうでしょう？「最初にテストコードから書き始めよう」なんて考える人は少ないでしょう。

JUnitを使えば、作成したプログラムをテストする為のコードを手早く書けますし、先にいくつかのテストコードを一旦用意しておけば、プログラム作成の作業を中断させることなく、早く、そして何回でもテストを実行できるようになります。

JUnitのテストはJavaで書かれていますので、ソースコードを書くこととテストを書くことの間に境目がなく、密接にからんでいます。

JUnit¥ Testing¥ Frameworkは、基本的に2つのデザインパターン(Command・Compositパターン)をキーとして設計されています。このフレームワークはテストを階層的に構成したり、テストの組み合わせを自動的に実行したりする環境を提供しています。ですから、個別のテストをひとくくりにし、他のテストの組と組み合わせるなど論理的なグループを１つのテストスイートとして構成できます。つまり、JUnitではテストを組み合わせてテストを実行することができます。

また、テストスイート自体を自動実行することができます。テストスイートを階層構

造にすることで、いろいろな論理レベルでテストを実行することができるようになります。

このようなテスト環境を提供してくれるJUnitですが、なんとフリーウェアです！ですから気軽に利用でき、テストにかかる費用の低コスト化を実現してくれます。

2.3. JUnit を使えば、品質の向上ができます

ここまで、JUnitを使うとテストが手早く、楽にできるということをお話しました。これでテストを何回でも実行できますね。ということは品質の向上にもつながります。

テストをすることで、コードの変更がソフトウェアのほかの部分に影響しないことを確認できます。つまり、ソフトウェアの構造的な整合性を保つことができます。
ブラックボックステスト(システム全体の動きを確認する)ではなく、ホワイトボックステスト(システム構成部品のそれぞれの中身をテストする)ですから、開発工程の中の小さな作業のひと区切りが完了した時に、それを検証するのに有効です。

品質を上げたいけれど、テストが面倒でコードに変更を加えることをためらう、なんて必要はもうありません。デバッグにかかる時間が減りますから、その分リファクタリングに時間を廻すことができます。

そう、JUnitは開発者向けのテストツールなのです。開発者の生産性とソースコードの品質の向上を目的として作られています。

3. JUnit インストール

JUnitのインストールの前提条件として、JDK1.3等のJava開発環境が既に整っていることが必要となります。当然のことですが、JUnitはJDKが必要になるからです。なお、JDKはSunのホームページから無償でダウンロードすることが可能です。

それではJUnitのインストール手順を説明していきます。JUnitのインストールは次のような流れになります。

1. まず、JUnitを入手します。JUnitはフリーソフトであるので、下記サイトよりダウンロードできます。
   http://www.junit.org/¥qquad (2001年5月27日現在 Version 3.7)
   ダウンロードするファイル¥quad JUnit3.7.zip
2. ダウンロードしたファイルを任意のディレクトリに解凍します。
3. 環境変数(CLASSPATH)を設定します。
   以下にWindows環境とLinux環境とにおいての設定方法の説明をします。
   - ＜Windows 環境の場合＞
     DOS プロンプトから SET コマンドを実行してください。
     またパスを永続的に設定するには、下記の手順を行ってください。
     - Windows 95/98 環境
       「メモ帳」等で、AUTOEXEC.BATファイルを開いて、次のようにCLASSPATH ステートメントを追加または変更します。
       CLASSPATH＜インストールディレクトリ＞￥JUNIT3.7￥juint.jar
     - Windows NT 環境
       「システムのプロパティ」の「環境」タブの「環境変数」にて、変数名を「CLASSPATH」とし、変数値は「＜インストールディレクトリ＞￥JUNIT3.7￥junit.jar」とし環境変数の登録を行ないます。
     - Windows 2000 環境
       「システムのプロパティ」の「詳細」タブの「環境変数」にて、変数名を「CLASSPATH」とし、変数値は「＜インストールディレクトリ＞￥JUNIT3.7￥junit.jar」とし環境変数の登録を行ないます。
   - ＜Linux 環境の場合＞
     - csh 環境
       ¥set path=($path ＜インストールディレクトリ＞/juint3.7/junit.jar)
     -
       PATH = $PATH: ＜インストールディレクトリ＞/juint3.7/junit.jar
4. ¥item 最後に JUnit を実行してテスト画面を起動して確認しましょう。

コマンドライン(Windows **環境の場合は、**DOS **プロンプト**)**より起動します。**

>java JUint.ui.TestRunner

図1**のような画面が表示されれば、インストール作業は完了です。**

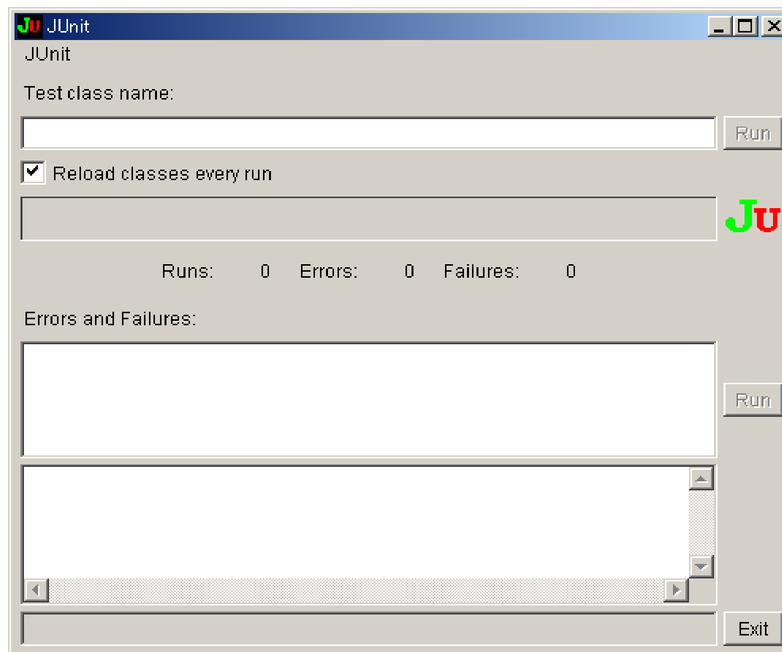


図 1 JUnit起動画面

4. JUnitの使い方

4.1. まず、使ってみよう

まずは、もっとも簡単なテストから述べてみましょう。JUnitに添付されているMoneyクラスの足し算のテストを例として、その手順を確認していきます。このテストは、同じ通貨2種類の金額の合計をテストする内容となっています。Moneyクラスは、引数に金額とその通貨を持ちます。CHFはスイスフランを表します。

このテストに必要な手順は、次のようになります。

1. TestCaseクラスを継承した、MoneyTestクラスを定義します。
2. コンストラクタ、main()メソッドを定義します。
3. テストしたい内容を記述したメソッドtestSimpleAdd()を定義します。
   12フランの値を持つ、Moneyクラスのインスタンス生成します。(f12CHF)
   14フランの値を持つ、Moneyクラスのインスタンス生成します。(f14CHF)
   足し算の後、期待される答え用のインスタンスを生成します。(expected)
   足し算の結果を表示するようにします。(assertEquals()メソッド)
4. コンパイルをし、テストを実行します。
5. Windows環境の場合は、MS-DOSプロンプトより次のようにタイプします。

```
javac¥ .¥junit¥samples¥money¥MoneyTest.java
java¥ junit.ui.TestRunnner¥ junit.samples.money.MoneyTest
```

以下がここで作成したテストコードです。

```
public void testSimpleAdd() {
			//インスタンスを作成します
			Money f12CHF;
			Money f14CHF;
			f12CHF= new Money(12, "CHF");
			f14CHF= new Money(14, "CHF");
			//期待される結果を用意します
			Money expected= new Money(26, "CHF");
			//12CHFインスタンスに14CHFインスタンスを加えます
			assertEquals(expected, f12CHF.add(f14CHF));
	}
```

4.2. テストの拡張

さきほどの足し算のテストに加え、引き算のテストを追加するとしたらどうなるのでしょうか。足し算のテストメソッドと同様のコードを用意し、足し算の操作を引き算の操作に変更したとします。確かにこの方法でもテストの実行は可能です。でもこれではテストを追加するたびに同じテストコードを繰り返すことになります。このような場合、複数のテストで重複する部分を共用することで解決できます。この共用するオブジェクトをフィクスチャー(fixture)と呼びます。フィクスチャーは簡単に記述することができますし、フィクスチャーを用いることでテストの労力を削減することができます。

引き算のテストを追加する場合のテスト手順は、次のようになります。

1. 12フラン、14フランをインスタンス変数にします
   Money f12CHF、Money f14CHFをインスタンス変数にするようコードを変更します。

```
==========================
    private    Money f12CHF;
    private    Money f14CHF;
==========================
```

2. フィクスチャーを初期化するメソッド(setUp())をオーバーライドします
   TestCaseクラスに予め用意されているsetUpメソッドをオーバーライドします。ここでは使用しませんが、フィクスチャーの後処理が必要な場合は、tearDown()メソッドをオーバーライドし、setUp()で作成したインスタンス変数を解放します。testSimpleAdd()の中でインスタンス生成している部分をsetUp()に移動します。このように一度フィクスチャーを宣言しておけば、多くのテストケースで利用できるようになります。

```
===============================
   protected void setUp() {
      f12CHF=¥ new Money(12, "CHF");
      f14CHF=¥ new Money(14, "CHF");
   }
===============================
```

3. 引き算用のテストメソッドを追加します
   引き算用のテストケースtestSimpleSubtract()を追加します。

ここまでで作成したサンプルテストコードは次のようになります。

```
package junit.samples.money;
import junit.framework.*;
public class MoneyTest extends TestCase {
	// define Money
	private	Money f12CHF;
	private	Money f14CHF;
	public MoneyTest(String name) {
		super(name);
	}
	public static void main(String args[]) {
		junit.textui.TestRunner.run(MoneyTest.class);
	}
	protected void setUp() {
		f12CHF= new Money(12, "CHF");
		f14CHF= new Money(14, "CHF");
	}
	public void testSimpleAdd() {
		// [12 CHF] + [14 CHF] == [26 CHF]
		Money expected= new Money(26, "CHF");
		assertEquals(expected, f12CHF.add(f14CHF));
	}
	public void testSimpleSubtract() {
		// [14 CHF] - [12 CHF] == [2 CHF]
		Money expected= new Money(2, "CHF");
		assertEquals(expected, f14CHF.subtract(f12CHF));
	}
}
```

Testの構造

TestCase MoneyTest	
import宣言	import junit.framework.*;
TestCase名を宣言	public MoneyTest(String name){super(name);}
Fixtureの宣言	private Money f12CHF; …
前処理	SetUp() f12CHF= new Money(12, "CHF"); ...
後処理	tearDown()
テスト対象1	testSimpleAdd()
テスト対象2	testSimpleSubtract()

suite()	public Static Test suite() {
	return new TestSuite(MoneyTest.class);
	}

mainの宣言	public static void main(String args[]){ String [] testCaseName = {MoneyTest.class.getName()};
TestRunner()	junit.ui.TestRunner.main(testCaseName)

図 2　テストの構造

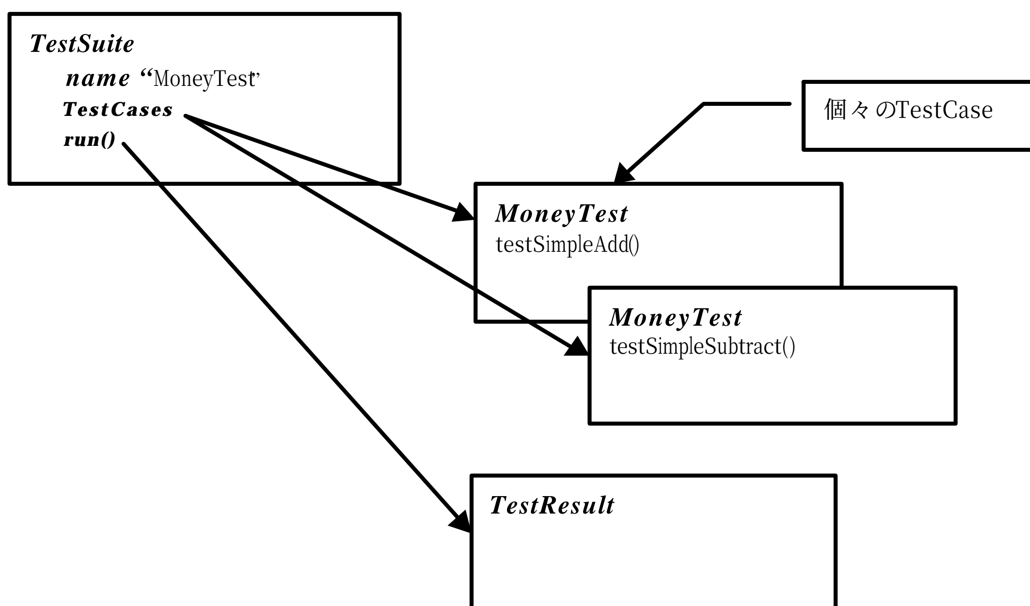


図 3　TestSuite の構造

4.3. TestRunner によるテストの自動実行と結果表示

JUnitは、作成したテストを自動的に実行する機能とその結果を表示する機能を提供しています。suite()メソッドに、テスト対象のメソッドを埋込む方法が例１です。この場合だと、テスト対象のメソッドが増えたらテストコードもどんどん増えていき、増えるたびに追加しなければなりません。これはたいへんです。そこで、テスト対象のメソッドを自動的に拾ってくれる機能があったら良いと思いませんか。JUnitでは、Javaのリフレクションという機能を利用して実現しています。テスト対象メソッドの名前をtestXXX()というように、"test"の４文字を先頭につけて、public で宣言してください。そして、suite()にtestXXX()を含んだクラスを渡してください。(例２参照) たったこれだけのコーディングで、対象のメソッドを自動実行してくれます。

例 1

```
public static Test suite() {
        TestSuite suite= new TestSuite() ;
        suite.addTest(new MoneyTest("testSubtract")) ;
        suite.addTest(new MoneyTest("testSimpleAdd")) ;
        return suite;
}
```

例 2

```
public static Test suite() {
        return new TestSuite(MoneyTest.class) ;
}
```

テストの起動

```
public static void main(String args[]) {
        junit.textui.TestRunner.run(suite()) ;
}
java MoneyTest
```

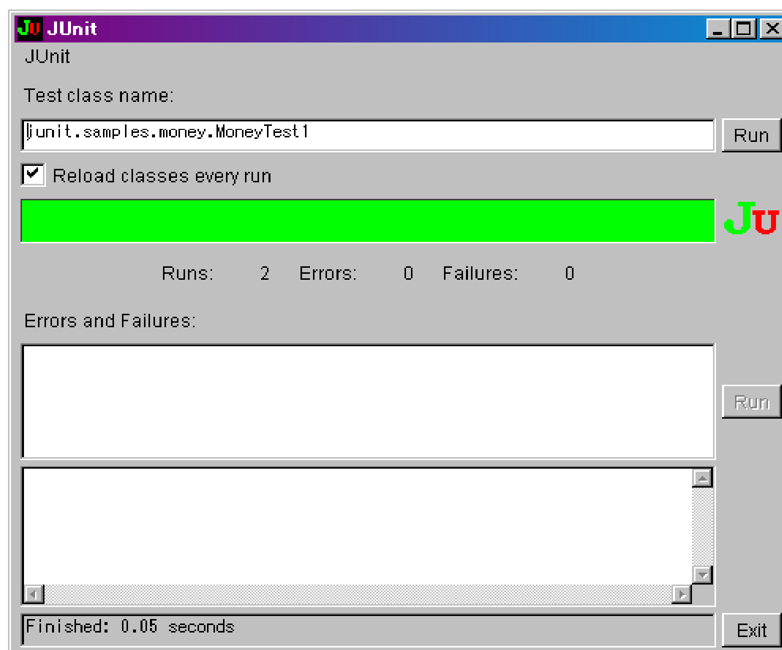


図 4 JUnit の実行結果(テスト成功)

JUnitは、TestRunnerの機能として、グラフィックス版とテキスト版の両方を提供します。

- java junit.textui.TestRunner テキストインタフェース
- java junit.ui.TestRunner GUIインタフェース
- java junit.awtui.TestRunner AWTインタフェース
- java junit.swingui.TestRunner Swingインタフェース

グラフィックスインタフェースでは、次のウィンドウを提供します。

- suite()メソッドをもつクラス名を入力するフィールド
- テストを起動する Run ボタン
- テストの進捗状況を示すインディケータ。テストが失敗したら緑色から赤色に変わる。
- ErrorsとFailuresの数が表示される
- 実行時間が表示される
- 失敗したテストの一覧が表示される
- スタックのトレースが表示される

JUnitでは、テストに成功しなかったら、ウィンドウの下の方に、失敗したテストの一覧を表示します。ここで、JUnitでは、Failures と Errors を区別します。Failuresの場

合は、期待した値とをチェックした結果が一致しなかった場合です。Errorsは、予期せぬ問題(例えば、ArrayIndexOutOfBoundsExceptionのような)が発生した場合です。図5は、失敗したテスト結果を表示した画面例です。Failure または Error について、さらに詳しく見たい場合は、対象のFailureまたは Error を選択して、Showボタンを押します。そうすると、その Failure または Errorのスタックのトレースが表示されます。

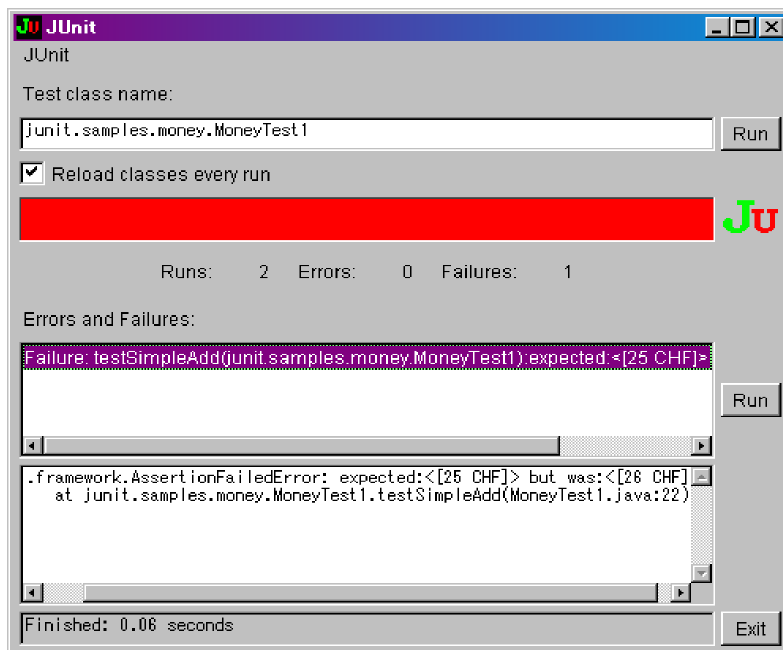


図 5 JUnit の実行結果(テスト失敗)

あとがき

今回の執筆作業において、執筆メンバが集まって打合せを行った時に、「リファクタリング」の言葉を機械翻訳したら「因数分解」という言葉になったと言う笑い話がありました。しかし、リファクタリングは単に、ソースをシンプルにすることではなく、設計や分析まで遡って行なうことです。そのことから見れば、「因数分解」という言葉は間違いとは言えないでしょう。まさに、「因数分解」のようにアプローチすることがリファクタリングには必要でしょう。

とは言え、ここでリファクタリングの説明をするわけではありません。

執筆メンバが何度となく、就業時間後に集まり、今回の執筆をまとめあげた作業そのものが、リファクタリングを行ったと言えるでしょう。彼らは、今回の執筆作業においてリファクタリングを体験し、少なくとも何かを感覚的に得たと思います。

最後に、彼らが英文のWebサイトに果敢に挑戦し、通勤時間や休日などの仕事以外の時間を費やして、この執筆をしてくれた努力に感謝いたします。

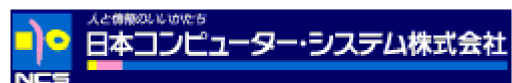
2001年7月

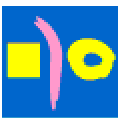
新保 康夫

平成13年7月4日 初版
監修 新保 康夫
著者 NCS XP研究会
高森 正延
岡田 将
大中 敏行
東 成樹
山中 美智子

許可なく、全部もしくは一部の複写・転載を禁ず。

Copyright © 2001¥ NCS¥ XP研究会.
Copyright © 2001¥ NCS¥ XP¥ Study Meeting.





NCS(http://www.ncs.co.jp/)