JUnit 4 の現状を紹介する

今度のリリースでは確実にテストが進化

Elliotte Rusty Harold
Adjunct Professor
Polytechnic University

2005年 9月 13日

JUnit は、Java 言語用のユニット・テスト・ライブラリーのデファクトスタンダードです。JUnit 4 は、このライブラリーにとって、過去約 3 年間で最初の重要リリースです。JUnit 4 では、テストの識別に関してサブクラス化やリフレクション (reflection)、命名規則 (naming convention) などに頼るのではなく、Java 5 の注釈機能を利用しており、それによってテストが単純化されると言われています。この記事では、コード・テストの偏執狂、Elliotte Harold が JUnit 4 を取り上げ、この新しいフレームワークを皆さんの作業の中でどのように使用するかについて解説します。この記事は、既に JUnit に経験のある読者を対象にしています。

JUnit は Kent Beck と Erich Gamma によって開発されたものですが、これまで開発されたサードパーティーの Java ライブラリーの中で、最も重要なものであることは疑いないでしょう。Martin Fowler はかつて、「ソフトウェア開発の世界で、これほど僅かのコード行で済ませられることに対して、これほど多くの人に感謝されているものは他に無いでしょう」と言っています。JUnit によって急激にテストが始まり、爆発的なテストの増加につながったのでした。JUnit のおかげで、それまでに比べて Java コードはずっと堅牢になり、信頼性が高まり、またバグ無しになりました。JUnit (Smalltalk の SUnit をもじったものです) から、様々な xUnit ツール・ファミリーができ、それによって広範な種類の言語にユニット・テストの恩恵が広まりました。つまり、nUnit (.NET)、pyUnit (Python)、CppUnit (C++)、dUnit (Delphi) などが、様々なプラットフォームや言語の世界で、無数のプログラマーをテスト病に感染させたのです。

しかし、JUnit は単なるテストにすぎません。JUnit による実際の恩恵は、フレームワーク自体にあるわけではなく、JUnit の中に具現化されている考え方や手法の中にあります。ユニット・テストや、テスト優先プログラミング (test-first programming)、テスト主導型開発などは、必ずしも JUnit で行う必要はありません。これは、GUI プログラミングは必ずしも Swing で行う必要がないのと同じことです。JUnit そのものが最後に更新されたのは、約 3 年前です。確かに JUnit は他の大部分のフレームワークよりも堅牢なことが知られており、また長く続いていますが、やはりバグは見つかっています。さらに重要なこととして、Java が進歩しているのです。Java は今や、ジェネリックスや列挙、可変長引数リスト、そして注釈をサポートしており、こうした機能によって、再利用可能フレームワークの設計に新しい可能性が生まれています。

JUnit の進行停止は、王座としての JUnit の地位を奪おうとするプログラマーの格好の標的となっていました。挑戦者としては、Bill Venners による Artima SuiteRunner から Cedric Beust による TestNG まで、様々なものがあります。これらのライブラリーは、推薦に値すべき機能を幾つか持っていますが、どれも JUnit が達成しているほどの精神や市場シェアには到達していません。どれも、そのままの形では、Ant や Maven、Eclipse のような幅広い製品をサポートする機能は持っていません。そこで Beck と Gamma は、Java 5 の新機能 (特に注釈) を活用して、オリジナル版の JUnit よりもユニット・テストをさらに単純化できるような、新しいバージョンの JUnit を作る作業に取りかかりました。Beckによると、「JUnit 4 のテーマは、さらに JUnit を単純化することによって、より多くの開発者が、より多くのテストを書くように仕向けること」です。JUnit 4 は既存の JUnit 3.8 テスト・スイートと後方互換性を維持していますが、Java のユニット・テストにおいて、JUnit 1.0 以来最も重要な革新となるはずです。

注意: このフレームワークで行われている変更は、未完成な部分が数多くあります。JUnit 4 の概要は明確になっていますが、詳細はまだ変更される可能性があります。この記事は JUnit 4 に関する暫定報告であって、最終報告ではありません。

テスト方法

JUnit のこれまでのバージョンではすべて、テストを見つけるために命名規則とリフレクションを使っています。例えば下記のコードは、1+1が 2つであることをテストしています。

```
import junit.framework.TestCase;

public class AdditionTest extends TestCase {

  private int x = 1;
  private int y = 1;

  public void testAddition() {
    int z = x + y;
    assertEquals(2, z);
  }

}
```

JUnit 4 は、これとは対照的に、テストは @Test 注釈で識別されます。これを下記に示します。

```
import org.junit.Test;
import junit.framework.TestCase;

public class AdditionTest extends TestCase {

  private int x = 1;
  private int y = 1;

  @Test public void testAddition() {
    int z = x + y;
    assertEquals(2, z);
  }

}
```

注釈を使う利点は、testFoo() や testBar() などのメソッドすべてに名前をつける必要がなくなることです。例えば、次のような手法でも動作するのです。

```
import org.junit.Test;
import junit.framework.TestCase;

public class AdditionTest extends TestCase {

  private int x = 1;
  private int y = 1;

  @Test public void additionTest() {
    int z = x + y;
    assertEquals(2, z);
  }

}
```

また、こうした方法でも動作します。

```
import org.junit.Test;
import junit.framework.TestCase;

public class AdditionTest extends TestCase {

  private int x = 1;
  private int y = 1;

  @Test public void addition() {
    int z = x + y;
    assertEquals(2, z);
  }

}
```

これによって、自分のアプリケーションに最も適切な命名規則に従えるようになります。例えば、私がこれまでに見た幾つかのサンプルでは、テスト・クラスがテスト・メソッドの名前に、テスト対象クラスと同じ名前を使う命名規則を使っています。例えば、List.contains() は ListTest.contains() によってテストされ、List.add() は ListTest.addAll() によってテストされ、などです。

TestCase クラスは相変わらず動作しますが、これを拡張することは要求されません。テスト・メソッドに @Test と注釈付けさえすれば、任意のクラスに、そのテスト・メソッドを入れられるのです。ただし、様々な assert メソッドにアクセスするためには、junit.Assert クラスをインポートする必要があります。これを次に示します。

```
import org.junit.Assert;

public class AdditionTest {

  private int x = 1;
  private int y = 1;

  @Test public void addition() {
    int z = x + y;
    Assert.assertEquals(2, z);
  }

}
```

また、JDK 5 での新しい静的インポート機能を使えば、これを古いバージョンの場合と同じくらい単純にすることができます。

```
import static org.junit.Assert.assertEquals;

public class AdditionTest {

  private int x = 1;
  private int y = 1;

  @Test public void addition() {
    int z = x + y;
    assertEquals(2, z);
  }

}
```

この方法によると、保護されたメソッドを含むクラスをテストケース・クラスが拡張できるため、メソッドをテストから守るのが容易になります。

SetUp と TearDown

JUnit 3 テスト・ランナーは、各テストを実行する前に自動的に setUp() メソッドを呼び出します。このメソッドは通常、フィールドを初期化し、ログをオンし、環境変数をリセットし、などを行います。例えば下記は、XOM の XSLTransformTest の setUp() メソッドです。

```
protected void setUp() {

    System.setErr(new PrintStream(new ByteArrayOutputStream()));

    inputDir = new File("data");
    inputDir = new File(inputDir, "xslt");
    inputDir = new File(inputDir, "input");

}
```

JUnit 4 でも、各テスト・メソッドを実行する前にフィールドを初期化でき、環境を設定できます。しかし、それを行うメソッドを setUp() と呼ぶ必要はもうありません。ただ単に @Before 注釈を付けた名前であれば良いのです。これを下記に示します。

```
@Before protected void initialize() {

    System.setErr(new PrintStream(new ByteArrayOutputStream()));

    inputDir = new File("data");
    inputDir = new File(inputDir, "xslt");
    inputDir = new File(inputDir, "input");

}
```

さらに、複数のメソッドに @Before 注記を付け、それぞれを各テストの前に実行するようにすることもできます。

```
@Before protected void findTestDataDirectory() {
    inputDir = new File("data");
    inputDir = new File(inputDir, "xslt");
    inputDir = new File(inputDir, "input");
}

 @Before protected void redirectStderr() {
    System.setErr(new PrintStream(new ByteArrayOutputStream()));
}
```

クリーンアップも同様です。JUnit 3 では、次のような tearDown() メソッドを使います (これは XOM で、大量のメモリーを消費するテストを行うような場合に使います)。

```
protected void tearDown() {
  doc = null;
  System.gc();
}
```

JUnit 4 では、より自然な名前をつけられ、それに @After という注釈を付けられるのです。

```
@After protected void disposeDocument() {
  doc = null;
  System.gc();
}
```

@Before の場合と同様、複数のクリーンアップ・メソッドに @After 注釈を付け、それぞれを各テストの後に実行するようにすることもできます。

最後に、スーパークラスにある初期化やクリーンアップのメソッドを明示的に呼ぶ必要は、もうありません。これらのメソッドがオーバーライドされていない限り、テスト・ランナーは必要に応じて、これらを自動的に呼んでくれるのです。スーパークラスにある @Before メソッドは、サブクラスにある @Before メソッドよりも前に呼び出されます。(これはコンストラクター呼び出しの順序と反対です。) @After メソッドの実行は、これとは逆です。サブクラスのメソッドはスーパークラスのメソッドの前に呼び出されます。そうしないと、@Before メソッドあるいは @After メソッドが複数ある場合の相対的な順序は保証されません。

スイート全体に渡る初期化

JUnit 4 では、JUnit 3 には無い、新しい機能も導入されています。つまりクラス・スコープを持った setUp() メソッドと tearDown() メソッドです。@BeforeClass 注釈が付いたメソッドはすべて、そのクラスの中にあるテスト・メソッドが実行される前に、必ず1度だけ実行されます。そして、@AfterClass 注釈の付いたメソッドはすべて、そのクラスの中にあるテストがすべて実行された後、必ず1度だけ実行されます。

例えば、そのクラスの中にある各テストが、初期化や廃棄にコストがかかる、データベース接続や、ネットワーク接続、非常に大きなデータ構造、あるいはその他のリソースを使う場合を考えてみてください。それらをテストの度に作り直す代わりに、1度だけ作り、1度で廃棄することができるのです。この方法を利用すると、一部のテストケースはずっと早く実行するようになります。例えば、サードパーティーのライブラリーの中にコールを行うエラー処理コードをテストする場合、私はよく、テストが始まる前に System.err をリダイレクトし、想定されるエラー・メッ

セージで出力が汚くならないようにします。そしてテストの終了後に、それを回復するのです。これを下記に示します。

```
// This class tests a lot of error conditions, which
// Xalan annoyingly logs to System.err. This hides System.err
// before each test and restores it after each test.
private PrintStream systemErr;

@BeforeClass protected void redirectStderr() {
    systemErr = System.err; // Hold on to the original value
    System.setErr(new PrintStream(new ByteArrayOutputStream()));
}

@AfterClass protected void tearDown() {
    // restore the original value
    System.setErr(systemErr);
}
```

テストの度に、テストの前後でこれを行う必要はありません。ただし、この機能には注意してください。この機能には、テストの独立性に違反し、予期しない結合を呼ぶ可能性があるのです。もし、あるテストが、@BeforeClass が初期化したオブジェクトを何らかの原因で変更すると、他のテストの結果に影響を与える可能性があります。テスト・スイートの中に順序に関する依存性が発生し、バグを隠してしまうかも知れません。どんな最適化でも同じですが、プロファイリングやベンチマークの結果、本当に問題があると分かってから、これを実装すべきです。とはいえ私は、あまりに実行に時間がかかるため、必要な回数だけ実行できないテスト・スイート (特に、数多くのネットワーク接続やデータベース接続が必要なもの) を 1つならず見てきています。(例えば、LimeWare テスト・スイートは実行に 2時間以上かかります。) こうしたテスト・スイートをスピードアップでき、プログラマーがもっと頻繁にテストを実行するように仕向けられるものであれば、どんなものでもバグの削減につながるはずです。

例外をテストする

例外のテストは、JUnit 4 で行われた最大の改善の 1つです。古いスタイルでの例外テストでは、例外を投げるコードの周囲にある try ブロックをラップし、その try ブロックの最後に fail() ステートメントを追加します。例えば、このメソッドは、ゼロで割ると ArithmeticException を投げることをテストします。

```
public void testDivisionByZero() {

    try {
        int n = 2 / 0;
        fail("Divided by zero!");
    }
    catch (ArithmeticException success) {
        assertNotNull(success.getMessage());
    }

}
```

このメソッドは醜いばかりではなく、(テストがパスするにせよ失敗するにせよ) 一部のコードが実行されないため、コード・カバレージ・ツールが失敗しがちなのです。JUnit 4 では、例外を投げるコードを書くことができ、また注釈を使って、例外が想定されていることを宣言できるようになっています。

```
@Test(expected=ArithmeticException.class)
  public void divideByZero() {
    int n = 2 / 0;
}
```

もし例外が投げられないと（あるいは別の例外が投げられると）、テストはフェールします。ただし、例外に関する詳細メッセージや他のプロパティーが必要な場合には、相変わらず、古い try-catch スタイルを使う必要があります。

無視されるテスト

皆さんの中には、実行に異常に時間がかかるテストを持っている人がいるかも知れません。そのテストにはもっと速く実行して欲しいものですが、そのテストが行っていることが、基本的に複雑、あるいは遅いのです。リモートのネットワーク・サーバーにアクセスするようなテストは多くの場合、このカテゴリーに入ります。もし皆さんが、そのテストを止めてしまう可能性のある何かに関して作業しているのでなければ、長時間実行するテスト・メソッドをスキップし、『コンパイル、テスト、デバッグ』のサイクルをスピードアップしたいと思うでしょう。また、皆さんが制御できる範囲外の理由で、テストがフェールしている場合もあるでしょう。例えば、W3C の XInclude テスト・スイートは、まだ Java がサポートしていない Unicode エンコーディングの幾つかに対する自動認識をテストします。こうした場合、赤いバー記号を無理に眺める代わりに、そうしたテストに @Ignore 注釈を付けてしまえばよいのです。これを次に示します。

```
// Java doesn't yet support
// the UTF-32BE and UTF32LE encodings
    @Ignore public void testUTF32BE()
      throws ParsingException, IOException, XIncludeException {

        File input = new File(
          "data/xinclude/input/UTF32BE.xml"
        );
        Document doc = builder.build(input);
        Document result = XIncluder.resolve(doc);
        Document expectedResult = builder.build(
          new File(outputDir, "UTF32BE.xml")
        );
        assertEquals(expectedResult, result);

    }
```

テスト・ランナーは、こうしたテストを実行せず、テストがスキップされたことを示します。例えば、テキスト・インターフェースを使うと、テストをパスしたことを示すピリオドや、テストにフェールしたことを示す「E」の代わりに、「I」 (ignore を表します) が出力されます。

```
$ java -classpath .:junit.jar org.junit.runner.JUnitCore
  nu.xom.tests.XIncludeTest
JUnit version 4.0rc1
.....I..
Time: 1.149

OK (7 tests)
```

ただし、よく注意してください。そもそもテストが書かれたのには、何らかの理由があるはずです。もし、そのテストを永遠に無視してしまうと、それらがテストするはずであったコードが動

作不良になるかもしれず、しかもその動作不良が、検出されないかも知れません。テストを無視することは、一時的な間に合わせであり、問題に対する真のソリューションではありません。

時間指定のテスト

パフォーマンスのテストは、ユニット・テストの中で最も面倒な領域です。JUnit 4 もこの問題を完全解決はしていませんが、前進が図られています。テストにはタイムアウト・パラメーターの注釈を付けることができます。そのテストの実行に、もし規定されたミリ秒以上かかる場合は、テストはフェールします。例えば次のテストは、その前にフィクスチャー (fixture) の中で設定された、文書中の全要素を見つけるのに 0.5秒以上かかると、フェールします。

```
@Test(timeout=500) public void retrieveAllElementsInDocument() {
    doc.query("//*");
}
```

ネットワークのテストには、(単純なベンチマーキングの他に) 時間指定のテストも有効です。例えば、あるテストが接続しようとしているリモート・ホストやデータベースがダウンしている場合、あるいは遅い場合、他のテストをホールド状態にしないように、そのテストをバイパスすることができます。また、良質なテスト・スイートは充分速く実行するものであり、プログラマーは大きな変更の後には毎回必ず、一日に何十回もそうしたテストを実行することができます。タイムアウトが設定できると、これが、より現実的になります。例えば次のテストは、http://www.ibiblio.org/xml の構文解析に2秒以上かかると、タイムアウトします。

```
@Test(timeout=2000)
  public void remoteBaseRelativeResolutionWithDirectory()
   throws IOException, ParsingException {
      builder.build("http://www.ibiblio.org/xml");
  }
```

新しいアサーション (assertion)

JUnit 4 では、配列の比較用に、2つの assert() メソッドが追加されています。

```
public static void assertEquals(Object[] expected, Object[] actual)
public static void assertEquals(String message, Object[] expected,
Object[] actual)
```

これらのメソッドは、配列の比較を最も明白な方法で行います。2つの配列が、同じ長さを持ち、それぞれの要素が、相手の配列で対応する要素と同じであれば、その 2つの配列は同じです。それ以外の場合は、2つの配列は異なります。一方、あるいは両方の配列がヌルである場合も、同じように処理されます。

足りないもの

JUnit 4 は根本的に新しいフレームワークであり、古いフレームワークのアップグレード版ではありません。JUnit 3 に慣れた開発者は、JUnit 4 では無くなっているものが幾つかあると思うかも知れません。

無くなっているものとして、最も明白なのは、GUI テスト・ランナーでしょう。テストがパスした時に出る、心を和ませる緑色のバーや、テストがフェールした時に出る、心配をかき立てる赤

いバーを見たいのであれば、Eclipse のような、JUnit サポートを統合した IDE が必要です。JUnit 4 では、Swing テスト・ランナーも AWT テスト・ランナーも更新されておらず、バンドルもされていません。

次に驚くことは、フェール (アサート・メソッドによってチェックされる、予期されるエラー) と、エラー (例外によって示される、予期せぬエラー) の間に区別が無くなったことです。JUnit 3 テスト・ランナーは、相変わらずこれらのケースを区別できますが、JUnit 4 ランナーは区別できません。

最後に、JUnit 4 には、複数のテスト・クラスからテスト・スイートを構築する suite() メソッドがありません。代わりに、可変長引数リストを使って、テストの数を規定せずにテスト・ランナーに渡せるようになっています。

GUI テスト・ランナーが無くなったことに対して、私はあまり快く思っていません。しかし、その他の変更は、JUnit をさらに単純化するために有効だと思います。こうした点の説明のために、現在どれほどの量のドキュメンテーションや FAQ が書かれているかを考えてみてください。その後で、JUnit 4 ではそれらを全く説明しなくて良いのだと考えれば、その素晴らしさが分かるでしょう。

JUnit 4 をビルドし、実行する

現在、JUnit 4 のバイナリー・リリースはありません。新しいバージョンで実験してみたい場合には、SourceForge にある CVS リポジトリーをチェックする必要があります。分岐は、「Version4」(参考文献) です。注意すべき点として、ドキュメンテーションの大部分は更新されておらず、古い、3.x 流の方法を説明しています。JUnit 4 では、注釈やジェネリックスなど、Java 5 言語レベルの機能を頻繁に使用するため、JUnit 4 のコンパイルには Java 5 が必要です。

コマンドラインからテストを実行するための構文は、JUnit 3 から少し変わっており、今度は org.junit.runner.JUnitCore クラスを使うようになっています。

```
$ java -classpath .:junit.jar org.junit.runner.JUnitCore
  TestA TestB TestC...
JUnit version 4.0rc1

Time: 0.003

OK (0 tests)
```

互換性

Beck と Gamma は、非常な努力を行って、前方互換性と後方互換性の両方を維持しています。JUnit 4 テスト・ランナーは、何も変更しなくても JUnit 3 テストを実行できます。JUnit 4 テストでの場合と同じように、皆さんが実行したい各テストのクラス名として完全修飾されたものを、単純にテスト・ランナーに渡せば良いだけです。ランナーは、どのテスト・クラスがどのバージョンの JUnit に依存しているかを判断し、適切に呼び出しを行います。

後方互換性は少し面倒ですが、JUnit 3 テスト・ランナーでも JUnit 4 を実行することができます。Eclipse など、統合 JUnit サポートを備えたツールは更新を行わなくても JUnit 4 を処理できるため、これは重要です。JUnit 3 環境で JUnit 4 テストが実行できるようにするに

は、JUnit4TestAdapter の中に JUnit 4 テストをラップします。次のようなメソッドを JUnit 4 テスト・クラスに追加すれば充分なはずです。

```
public static junit.framework.Test suite() {
  return new JUnit4TestAdapter(AssertionTest.class);
}
```

ただし Java に関しては、JUnit 4 は全く後方互換性がありません。JUnit 4 は、完全に Java 5 の機能に依存しています。Java 1.4 以前のバージョンでは、コンパイルも実行もできません。

これから先は

JUnit 4 はまだ完成していません。ドキュメンテーションの大部分を含めて、幾つかの重要部分が欠けています。私としては、皆さんのテスト・スイートを注釈や JUnit 4 に変換してしまうことは、まだお勧めしません。しかし、開発は急ピッチで進んでおり、JUnit 4 は非常に大きなものを約束してくれそうです。Java 2 プログラマーは、しばらくの間 JUnit 3.8 にとどまるでしょうが、Java 5 に移行した人は、それに合わせて、自分たちのテスト・スイートをこの新しいフレームワークに適応させることを考え始めるでしょう。

著者について

Elliotte Rusty Harold

Elliotte Rusty Harold はニューオーリンズ出身であり、時たま、おいしい gumbo (オクラ入りのスープ) を食べに帰っています。ただし現在はニューヨークのブルックリン近郊の Prospect Heights に、妻の Beth と猫の Charm (charmed quark からとりました) と Marjorie (義理の母の名前からとりました) と一緒に住んでいます。彼は Polytechnic University のコンピューター・サイエンスの非常勤教授として、Java 技術とオブジェクト指向プログラミングを教えています。彼の Cafe au Lait Web サイトは、インターネット上で最も人気のある独立系 Java サイトの一つです。また、そこから派生した Cafe con Leche は、最も人気のある XML サイトの一つです。彼の最近の著作には『Java I/O, 2nd edition』があります。現在は XML 処理用の XOM API や Jaxen XPath エンジン、Jester テスト・カバレッジ・ツールなどに取り組んでいます。