TestNGでJavaユニット・テストを楽々行う

JUnit に勝るTestNGフレームワークを試す

Filippo Diotalevi
IT Specialist
IBM Italy

2005年 1月 06日

JUnit フレームワークはJava言語のユニット・テストとして、一つですべての役割を果たすことができます。Java開発においてテスト駆動開発(test-driven development)という概念をもたらしたこと、また効果的なユニット・テストの書き方を教えた、という点でJUnit フレームワークは称賛に値するものです。ところがこの数年、JUnitはごく僅かしか進化していません。JUnitは補完的な他のテスト・フレームワークと統合して使う必要があるため、今日のように複雑な環境に向けたテストを書くことが次第に難しくなっています。この記事では、Filippo Diotalevi が、Javaアプリケーションのテスト用の新しいフレームワークであるTestNGを紹介します。TestNGは非常に強力で革新的、拡張可能そして柔軟なだけではなく、JDK 5.0の素晴らしい新機能であるJavaアノテーション（ Annotations ）の応用例として、非常に興味深いものにもなっています。

最近のソフトウェア・パッケージではどれも、構築フェーズの中心をテストが占めるようになっています。まずはコーディング、そして時間のある時に（ 時間が無くても ）ちょっとばかりテストをするという、古き良き時代は過去のものとなりました。多くの開発者は今や、開発プロセスのごく初期段階でバグを捉え、また主なリスクを特定できるように、コーディングとテストを区別なく並行して行えるソフトウェア手法が必要だと感じています。

JUnitは他のどんなテスト・フレームワークにもまして、開発者がテスト、特にユニット・テストの重要性を認識するための推進力となりました。単純で実用的、そして厳密なアーキテクチャーを持つJUnitは、膨大な数の開発者を「熱中」させることができたのです。（ 「test infected(テスト熱中症)」に関しては、参考文献を見てください。 ）JUnitのユーザーはユニット・テストに関する基本的なルールとして、次のようなものを学びました。

- どんな小さなコードも、必ずテストする必要がある
- コードはできうる限り、他から分離した状態で（ 例えば疑似オブジェクト（ mock objects ）のような手法を使って ）テストする必要がある
- ソフトウェアはテストが容易である必要がある・・・つまりテストを念頭にソフトウェアを書く必要がある

ところが開発者がテストに自信を深めるにつれ、JUnitの持つ単純さと厳密さに関して、まったく逆の見方がされるようになりました。一方の開発者グループは、JUnitの持つ単純さこそが、ソフトウェアも単純であるべきだと常に意識する元になると主張します（これはKISS原則として知られています。KISS:keep it simple, stupid(単純にしておけ、アホ、の意)）。他方、JUnitは単純なのではなく安易であるとするグループもあり、新しい高度機能、さらなる柔軟性、さらなる強力さを求めています。JUnitの問題点として、後者のグループから特に批判されているのは、次のような点です。

- Javaには単一継承しかないため、`TestCase`クラスを継承する必要があるが、これが大きな制限要素になっている
- パラメーターを、JUnitのテスト・メソッドや`setUp()`メソッド、`tearDown()`メソッドに渡すことができない
- 実行モデルが少しおかしい。テスト・メソッドが実行される度に、テスト・クラスが再インスタンス化される
- 複雑なプロジェクトにおいて、様々なテスト・スイートを管理するのが難しい

TestNGの作成者

TestNGを作ったのはCedric Beustです。Javaプログラミングではよく知られた名前ですが、EJB 3エキスパート・グループのメンバーであり、その他にもEJBGenやDoclipseなど一般的なオープン・ソース・プロジェクトも作っています。TestNGはApache Software Licenseの条件で配布されており、Webサイトからダウンロードすることができます（このサイトとCedricのサイトへのリンクは参考文献にあります）。

この記事では、TestNGと呼ばれる新しいテスト・フレームワークを使ってアプリケーションのユニット・テストを書く方法について学びます。TestNGはJUnitの精神を継承したもので、JUnitの単純さを保ったままJUnitの持つ制限をほとんど取り払い、より柔軟で強力なテストが書けるようになっています。TestNGはテストの定義としてJava アノテーションに大きく依存しています。ですからTestNGを見ることによって、この新しいJava機能を、実稼働状態で見ることもできるのです。（Java アノテーションはJDK 5.0で導入されました。これについて詳しくは参考文献を見てください。）

コードについて

TestNGの使い方を説明するために、広く使われているオープン・ソース・ライブラリー用にユニット・テストを書くことにします。これはJakarta Common Langと呼ばれるライブラリーですが、文字列や数字、それにJavaオブジェクトなどを操作・処理するために便利なクラスを持っています。TestNGとJakarta Common Langライブラリーへのリンクは、参考文献にあります。皆さんが自分のマシンでこの記事と同じことをする場合には、両方ともダウンロードする必要があります。

TestNGを入手するには2つの異なるパッケージがあります。JDK 5.0が必要なものと、Javaのバージョン1.4で動作するものです。両者は、テスト定義の構文が、わずかに異なります。前者はJDK 5.0 アノテーションを使用し、後者は古い、Javadocスタイルのアノテーションを使います。この記事ではJDK 5.0用のものを使うので、この先を読み進むためにはアノテーションを基本的に理解している必要があります。アノテーションに関するdeveloperWorksの資料へのリンクは参考文献に挙げてあります。ただしJDK 5.0は、テストをコンパイルして実行するために必要なだけである、ということは重要ですので、よく理解してください。アプリケーション・コードをビルドするには、相変わらず自分の好きなコンパイラーが使えるのです。実際ここでは、Jakartaプロ

ジェクトのWebページからダウンロードできるのと同じJARファイルを使って、Jakarta Common Langライブラリーをテストするのです。バージョン1.4のJavaでTestNGを使うための詳細に関しては、TestNGのWebサイトに説明されています。

最後に、この記事の先頭と最後にあるCodeアイコンをクリックしてj-testng-sample.zipをダウンロードします。このファイルには、TestNGを使ってJakarta Commons Lang用にユニット・テストを書く方法の例が幾つか含まれています。また、この記事で紹介しているコードの大部分とは別に、幾つかのサンプルも含まれています。この記事を読むためにコードは必要ありませんが、コードを見た方が、ここで説明している概念をより深く理解できるでしょう。

TestNG早わかり

TestNGテスト・クラスは、単純に昔ながらのJavaオブジェクトです。テスト・メソッド用に特別なクラスを継承したり、特別な名前の付け方を使ったりする必要はありません。単にアノテーション`@Test`を使って、テストであるクラスのメソッドをフレームワークに対して知らせるのです。リスト1はユーティリティー・クラス`StringUtils`に対するテストとして、一番単純なものの一つです。このテストは`StringUtils`の2つのメソッドをテストします。`isEmpty()`は`String`が空かどうかをチェックし、`trim()`は`String`の前後から制御文字を除去します。エラー条件のチェックにJava命令、`assert`が使われていることに注意してください。

リスト1. StringUtilsクラスに対するテスト・ケース

```
package tests;
import com.beust.testng.annotations.*;
import org.apache.commons.lang.StringUtils;
public class StringUtilsTest
{
 @Test
 public void isEmpty()
 {
  assert StringUtils.isBlank(null);
  assert StringUtils.isBlank("");
 }
 @Test
 public void trim()
 {
  assert "foo".equals(StringUtils.trim(" foo   "));
 }
}
```

ただしテストを実行する前に、よくありそうなtestng.xmlという名前の特別なXMLファイルを使って、TestNGを設定する必要があります。このファイルの構文は、リスト2に示すように非常に単純です。このファイルはテスト・スイート、My test suiteを定義するところから始まります。My test suiteは、`StringUtilsTest`クラスによって作られるユニークなテストである、First testから構成されています。

リスト2. TestNG用のコンフィギュレーション・ファイル

```
<!DOCTYPE suite SYSTEM "http://beust.com/testng/testng-1.0.dtd" >
<suite name="My test suite">
  <test name="First test">
    <classes>
       <class name="tests.StringUtilsTest" />
    </classes>
  </test>
</suite>
```

このサンプル、testng.xmlファイルはあまり役に立つように思えないかも知れませんが（テスト・クラスは一つしかありません）、実は素晴らしいことに、このファイルを書きさえすれば、テスト・スイートを定義できるのです。JUnitの古き時代を思い出してください。JUnitの時代には、テスト・スイートの定義はたいていの場合、JUnitの`TestSuite`やプロパティ・ファイル、それに当然ながらAntビルド・ファイルなど、幾つかのファイルに広がっていたものです。TestNGでは、必要なデータはすべてtestng.xmlファイルに集められています。`TestSuite`はもはや無く、ビルド・ファイルもずっと小さいのです。

テストを実行するには、このクラスを`javac`でコンパイルしてから、次のコマンドでTestNGを呼び出します。

```
java -ea -classpath .;testng.jar;commons-lang-2.0.jar com.beust.testng.TestNG testng.xml
```

ここでオプション`-ea`は、JVMに対してアサーションを処理するように（そしてアサーションがフェールした場合には例外を上げるように）伝えます。この例を実行するのに必要なのは、testng.jarとcommons-lang-2.0.jarという2つのライブラリーのみであり、`com.beust.testng.TestNG`はTestNGのメイン・クラスです。のんきなことに`java`と`javac`の神秘な構文を忘れてしまった怠け者の開発者のために、便利なAntタスクも利用できます。リスト3は一例として、この記事で配布しているサンプル・アプリケーションのAntビルド・ファイルを示しています。クラス`com.beust.testng.TestNGAntTask`に関連付けられた`testng`タスクの定義と、testターゲットでの`testng`タスクの簡単な使い方に注意してください。

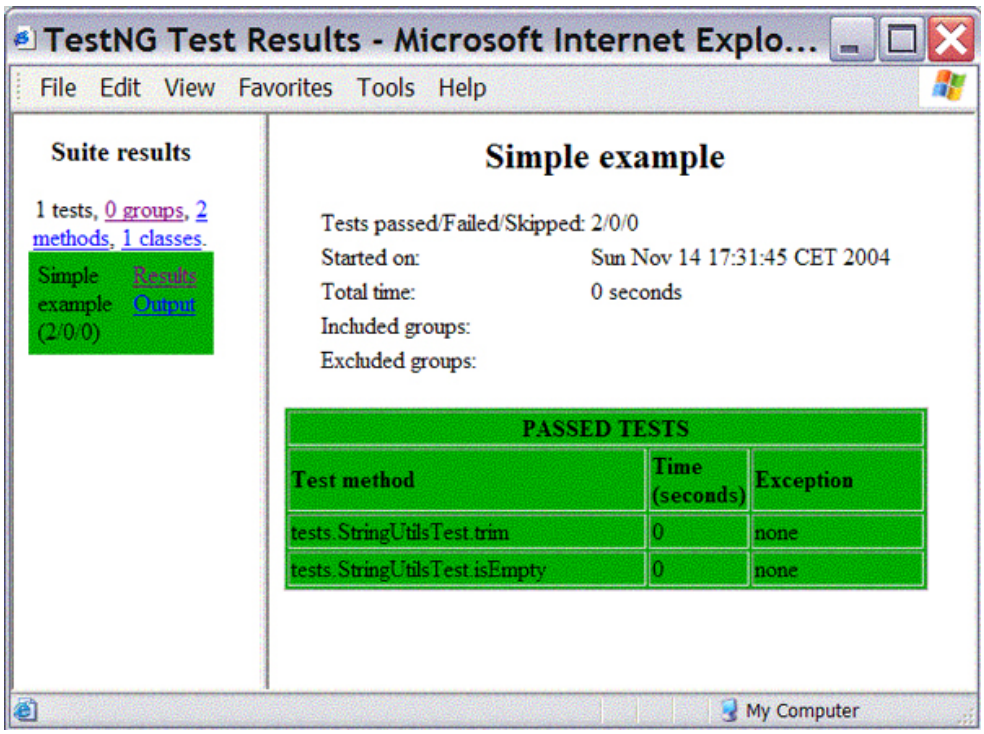
リスト3. TestNGタスクでのAntビルド・ファイル

```
<project name="sample" default="test" basedir=".">
   <!-- COMPILE TESTS-->
   <path id="cpath">
      <pathelement location="testng.jar"/>
      <pathelement location="commons-lang-2.0.jar"/>
   </path>
   <target name="compile">
      <echo message="compiling tests"/>
      <mkdir dir="classes"/>
      <javac   debug="true"
         source="1.5" classpathref="cpath"
         srcdir="src" destdir="classes"/>
   </target>
   <!-- RUN TESTS-->
   <taskdef name="testng"
      classname="com.beust.testng.TestNGAntTask"
      classpathref="cpath"/>
   <path id="runpath">
      <path refid="cpath"/>
      <pathelement location="classes"/>
   </path>
   <target name="test" depends="compile">
      <echo message="running tests"/>
      <testng fork="yes" classpathref="runpath" outputDir="test-output">
         <fileset dir="src" includes="testng.xml"/>
         <jvmarg value="-ea" />
      </testng>
   </target>
</project>
```

すべてが正しく行われれば、コンソールにテストの結果が表示されるはずです。さらにTestNGは、カレント・ディレクトリーに自動的に作られるtest-outputというフォルダーに、非常に素晴

らしいHTMLレポートを生成するのです。このファイルを開いてindex.htmlをロードすると、図1に示すようなページを見ることができます。

図1. TestNGによって生成されるHTMLレポート



テスト・グループを定義する

TestNGの機能としてもう一つ面白いのは、何グループものテストを定義できることです。どのテスト・メソッドも、一つ以上のグループに関連付けることができます。一旦こうしたグループを定義しておけば、あるグループのテストだけを実行するように選択することができるのです。テスト・グループにテストを追加するには、単に次の構文を使って、そのグループを`@Test`アノテーションのパラメーターとして規定します。

```
@Test(groups = {"tests.string"})
```

具体的にこの例では、アノテーションを付けられたメソッドが`tests.string`グループに属することを宣言しています。パラメーター`groups`は配列なので、それぞれのグループ名をカンマで区切ることで、複数のグループを規定することができます。例えばサンプル・アプリケーションでは、`String`や数字、ブール値などに対して別々のテストを作ってから、それらを選択的に実行するようにTestNGを設定することもできます（リスト4）。

リスト4. 様々なグループを持つコンフィギュレーション・ファイル

```
<!DOCTYPE suite SYSTEM "http://beust.com/testng/testng-1.0.dtd" >
<suite name="My suite">
  <test name="Simple example">
    <groups>
      <run>
  <include name="tests.string" />
  <include name="tests.math" />
  <exclude name="tests.boolean"/>
      </run>
    </groups>
    <classes>
  .... list classes here....
    </classes>
  </test>
</suite>
```

当然ですが、別々のテスト・グループを実行するとHTMLレポートは全テストを一つのリストに表示し、また各グループに対して別のリストを表示します。ですから、問題の原因を即座に理解することができます。

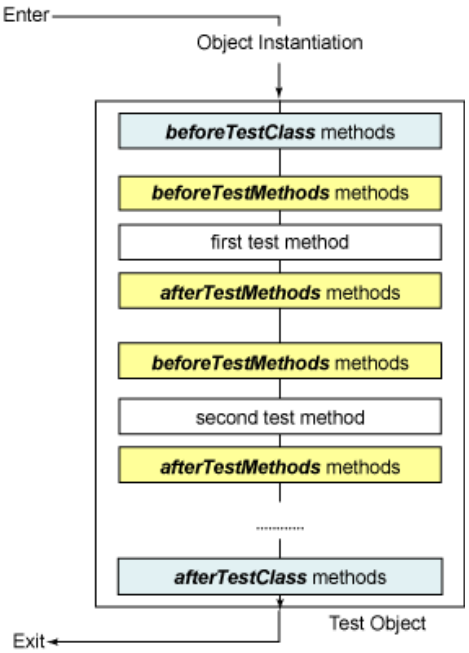
コンフィギュレーション・メソッド

TestNGで規定できるのはテスト方法だけではありません。コンフィギュレーション・メソッドという専用のアノテーション @Configurationを使って、クラス中の特定なメソッドを規定することもできるのです。コンフィギュレーション・メソッドには下記の4つのタイプがあります。

- `beforeTestClass`メソッドは、クラスがインスタンス化された後、まだどのテスト・メソッドも実行されていない時に実行される
- `afterTestClass`メソッドは、クラス中のテスト・メソッドがすべて実行された後に実行される
- `beforeTestMethod`メソッドは、クラス中の、どのテスト・メソッドよりも前に実行される
- `afterTestMethod`メソッドは、クラス中の各テスト・メソッドが実行される度に、そのメソッドの実行後に実行される

図2はテスト・クラスのライフサイクルを示しています。

図2. テスト・クラスのライフサイクル



リスト5はコンフィギュレーション・メソッドの例を幾つか示しています。グループを使う時には、コンフィギュレーション・メソッドもグループに属す必要があることに注意してください。また、4つのコンフィギュレーション・メソッドは相互排他的ではありません。ですから、この4つのうち2つ以上の範疇に同時に属するメソッドを定義することもできます（一例として、リスト5にある`aroundTestMethods()`メソッドを見てください）。

リスト5. コンフィギュレーション・メソッドの例

```
@Configuration(beforeTestClass = true, groups = {"tests.workflow"})
public void setUp()
{
   System.out.println("Initializing...");
}
@Configuration(afterTestMethod = true, beforeTestMethod = true, groups = {"tests.workflow"})
public void aroundTestMethods()
{
   System.out.println("Around Test");
}
```

TestNGのコンフィギュレーション・メソッドは、JUnitの`setUp()`メソッドや`tearDown()`メソッドよりも強力であり、テストの実行を適切にすることと、テスト・ケースの実行後にデータを更新することを主な目的としています。

例外チェック

TestNGでは、例外の発生を非常に単純かつ容易にチェックすることができます。もちろんこれはJUnitでも可能ですが、TestNGで`@ExpectedExceptions`アノテーションを使った方が、驚くほど単純かつ容易にテスト・コードが書けるようになります。この例をリスト6に示します。`@ExpectedExceptions`アノテーションは、TestNGフレームワークでは`NumberFormatException`を上げることが許されていることを規定しています。ですか

らNumberFormatExceptionが上がっても、失敗と見なしてはなりません。あるコード行で例外が上げられているかどうかを見るには、その行の直後にassert falseステートメントを追加します。これは、特定したそのタイプの例外が、その行で上げられた時にのみ、テストをパスすることを意味しています。

リスト6. TestNGでの例外チェック

```
public class  NumberUtilsTest
{
 @Test(groups = {"tests.math"})
 @ExpectedExceptions(NumberFormatException.class)
 public void test()
 {
  NumberUtils.createDouble("12.23.45");
  assert false;  //shouldn't be invoked
 }
}
```

まとめ

この記事では、皆さんがすぐにユニット・テストが書き始められるようになることを念頭に、TestNGを簡単かつ実用的に紹介しました。ただしこれは完全な参照マニュアルではありません。TestNGには他にも、非常に興味深く、かつ便利な機能が豊富に含まれています。

- テストやコンフィギュレーション・メソッドに対して引き数を渡すことができ、それをアノテーションで、あるいはXMLコンフィギュレーション・ファイルの中で宣言することができる
- 「互換モード（compatibility mode）」を使って、古き良きJUnitをTestNGの下で実行することができる
- テスト・グループ間で依存関係を確立できるので、実行の順序が決められる

TestNGの持つあらゆる可能性を理解するためには、そのドキュメンテーション（参考文献）を読む必要があるでしょう。

こうした機能に加え、テストの定義にJava アノテーションが採用されていることから、テストのプロセス全体が非常に単純かつ柔軟になります。テストを書くために守るべきルールは、ほんの少しだけです。それさえ守れば、自分で好きなようにテスト戦略を選択できるのです。

TestNGを使っていると、このフレームワークが既にユニット・テストを書くための選択肢として適切であることがよく分かります。つまり他のライブラリーやツールとの統合が容易なように設計されているのです。ですから今後TestNGがさらに発展して行けば、開発者にとって大いに興味をそそるものがもたらされるでしょう。新しい道が敷かれたのです。

ダウンロード

内容	ファイル名	サイズ
j-testng-sample.zip	j-testng-sample.zip	

著者について

Filippo Diotalevi

Filippo DiotaleviはミラノにあるIBM ItalyのITスペシャリスト（IT専門家）であり、そこでは主にJ2EEアプリケーション開発者として勤務します。彼が関心を抱く分野は、パターン、アスペクト指向プログラミング、そしてアジャイル方法論です。IBM Redbook「Patterns: Direct Connections for Intra- and Inter-enterprise」の共著者、Webサイトそして雑誌で発行された数多くの技術記事の著者、そしてJava User Group Milanoの創立者でもあります。