コード品質を追求する: JUnit 4 対 TestNG

大規模なテストでは TestNG のほうが優れたフレームワークになる理由

Andrew Glover
CTO
Stelligent Incorporated

2006年 8月 29日

新たなアノテーション・ベースのフレームワーク、JUnit 4 には TestNG の選り抜きの機能がいくつか含まれていますが、それによって TestNG が使われなくなることを意味するのでしょうか。この記事では、Andrew Glover がそれぞれのフレームワークの特徴を検討し、まだ TestNG にしか見当たらない高度な 3 つのテスト機能を明らかにします。

JUnit 4.0 は、長い間開発が中断された末、今年の始めにリリースされました。JUnit フレームワークの変更内容のなかでも特にこの記事の読者にとって注目に値する変更は、アノテーションを賢く使うことによって実現されています。ルック・アンド・フィールが大幅に更新されただけでなく、新しいフレームワーク機能により、テスト・ケースを作成する際の構造的規則が劇的に緩和されています。以前は融通が利かなかったフィクスチャー・モデルにも柔軟性が加わり、一層構成しやすくなっています。その結果、JUnit ではテストを test で始まる名前のメソッドとして定義する必要がなくなり、フィクスチャーを各テストに対して一回だけ実行すればいいようになりました。

このような変更は大いに歓迎できますが、アノテーション・ベースの柔軟なモデルを提供する Java™ テスト・フレームワークは JUnit 4 が初めてではありません。JUnit の修正が行われるかなり以前に、TestNG はアノテーション・ベースのフレームワークとして、その地位を確立していました。

実際、TestNG は Java プログラミングでのアノテーションを用いたテスト方法の先駆けであり、それ故に JUnit の恐るべきライバルとなっていますが、JUnit 4 がリリースされて以来、多くの開発者がこの 2 つのフレームワークに違いがあるのか疑問に思っています。そこで、今月の記事では TestNG を JUnit 4 から分け隔てている機能を説明し、2 つのフレームワークを争わせるのではなく、相補し続けていく方法を提案します。

ご存知でしたか?

Ant では、JUnit 4 テストを実行するのが予想以上に困難であることが判明しています。ちなみに一部のチームでは、Ant 1.7 へのアップグレードが唯一のソリューションであると結論しています。

うわべはよく似ています

JUnit 4 と TestNG には、共通するいくつかの重要な特性があります。どちらもテストを驚くほど単純に (しかも面白く) するフレームワークで、活気のあるコミュニティーによって積極的な開発がサポートされると同時に、豊富なドキュメンテーションも用意されています。

コードの品質向上

急を要する質問がある場合は、Andrew のディスカッション・フォーラムで答えを探してください。

2 つのフレームワークの違いは、そのコア設計にあります。JUnit はユニット・テスト用のフレームワークです。つまり単一のオブジェクトをテストしやすくすることを目的にビルドされているため、単一オブジェクトのテストを極めて効率的に処理します。一方 TestNG は、それよりも高位レベルでのテストに対応するようビルドされているため、JUnit にはない機能が用意されています。

単純なテスト・ケース

JUnit 4 と TestNG にそれぞれインプリメントされたテストは、一見すると非常によく似ています。これを確かめるには、リスト 1 のコードを見てください。このコードは JUnit 4 のテストで、@BeforeClass 属性で示されたマクロ・フィクスチャー (テストが実行される前に一度だけ呼び出されるフィクスチャー) があります。

リスト 1. 単純な JUnit 4 のテスト・ケース

```
package test.com.acme.dona.dep;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import org.junit.BeforeClass;
import org.junit.Test;

public class DependencyFinderTest {
 private static DependencyFinder finder;

 @BeforeClass
 public static void init() throws Exception {
  finder = new DependencyFinder();
 }

 @Test
 public void verifyDependencies()
  throws Exception {
   String targetClss =
     "test.com.acme.dona.dep.DependencyFind";

   Filter[] filtr = new Filter[] {
      new RegexPackageFilter("java|junit|org")};

   Dependency[] deps =
      finder.findDependencies(targetClss, filtr);

   assertNotNull("deps was null", deps);
   assertEquals("should be 5 large", 5, deps.length);
 }
}
```

JUnit ユーザーならすぐに、このクラスには以前の JUnit バージョンで必要だったシンタックス・シュガーのほとんとがなくなっていることに気付くはずです。setUp() メソッドは見当たらず、このクラスは TestCase を拡張しないだけでなく、test で始まるメソッドもありません。このクラスはまた、静的インポートや、そしてもちろんアノテーションなどの Java 5 機能も利用しています。

さらに高まる柔軟性

リスト 2 に、同じテストを今度は TestNG を使ってインプリメントした場合を示します。このコードとリスト 1 のテストにはわずかな違いがありますが、それが何だかわかりますか。

リスト 2. TestNG のテスト・ケース

```
package test.com.acme.dona.dep;

import static org.testng.Assert.assertEquals;
import static org.testng.Assert.assertNotNull;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.Configuration;
import org.testng.annotations.Test;

public class DependencyFinderTest {
 private DependencyFinder finder;

 @BeforeClass
 private void init(){
  this.finder = new DependencyFinder();
 }

 @Test
 public void verifyDependencies()
  throws Exception {
   String targetClss =
     "test.com.acme.dona.dep.DependencyFind";

   Filter[] filtr = new Filter[] {
      new RegexPackageFilter("java|junit|org")};

   Dependency[] deps =
      finder.findDependencies(targetClss, filtr);

   assertNotNull(deps, "deps was null" );
   assertEquals(5, deps.length, "should be 5 large");
 }
}
```

上記の 2 つのリストは一見すると非常によく似ていますが、注意して見ると、JUnit 4 よりも TestNG のコーディング規則のほうが柔軟であることが分かります。リスト 1 の JUnit では、@BeforeClass 修飾メソッドを static メソッドとして宣言する必要があったため、フィクスチャーである finder も static として宣言しなければなりませんでした。また、init() メソッドを public として宣言する必要もありました。リスト 2 を見てみると、話は別であることが分かります。TestNG では、このような規則が必要ないからです。そのため、init() メソッドは static でも public でもありません。

柔軟性は当初から、TestNG の長所の一つでしたが、セールス・ポイントはそれだけではありません。TestNG には、JUnit 4 にはないテスト機能も備わっています。

依存関係のテスト

JUnit フレームワークが実現を目指していることとして、テストの分離があります。これは逆に、テスト・ケースの実行順を指定するのを困難にしますが、テスト・ケースの実行順は依存関係のテストにとって非常に重要です。この問題を回避するため、開発者はさまざまな手法を使用しています。例えば、テスト・ケースをアルファベット順にする、フィクスチャーに重点を置いて正しくセットアップするなどです。

このような回避方法は、テストに合格すればいいのですが、テストに失敗した場合、後続の依存関係のテストもすべて失敗という結果になってしまいます。場合によっては、大規模なテスト・スイートで、不要な失敗までレポートすることにもなりかねません。例えば、ログインが必要な Web アプリケーションをテストするテスト・スイートがあるとします。この場合、JUnit の分離主義に対処する方法として、テスト・スイート全体をアプリケーションへのログインによってセットアップする従属メソッドを作成することが考えられます。これは賢い解決方法ですが、ログインに失敗した場合、アプリケーションの事後ログイン機能が働くとしてもテスト・スイート全体が失敗してしまいます。

失敗にするのではくスキップする

JUnit とは異なり、TestNG では Test アノテーションの dependsOnMethods 属性によってテストの依存関係を難なく処理します。この便利な機能を使用すれば、目的のメソッドの前に実行する従属メソッド (上記のログインなど) を簡単に指定することができます。さらに、従属メソッドが失敗した場合、すべての後続テストは失敗としてマークされるのではなく、スキップされることになります。

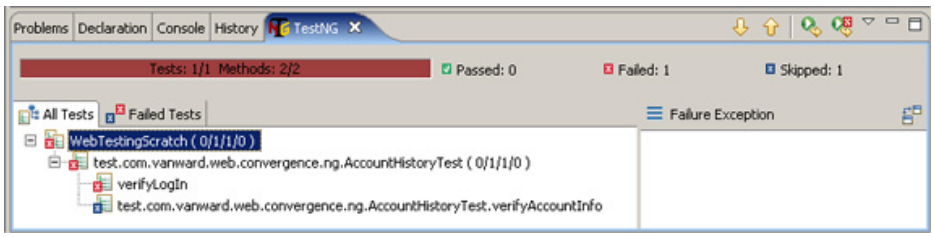
リスト 3. TestNG による依存関係のテスト

```
import net.sourceforge.jwebunit.WebTester;

public class AccountHistoryTest  {
 private WebTester tester;

 @BeforeClass
 protected void init() throws Exception {
  this.tester = new WebTester();
  this.tester.getTestContext().
   setBaseUrl("http://div.acme.com:8185/ceg/");
 }

 @Test
 public void verifyLogIn() {
  this.tester.beginAt("/");
  this.tester.setFormElement("username", "admin");
  this.tester.setFormElement("password", "admin");
  this.tester.submit();
  this.tester.assertTextPresent("Logged in as admin");
 }

 @Test (dependsOnMethods = {"verifyLogIn"})
 public void verifyAccountInfo() {
  this.tester.clickLinkWithText("History", 0);
  this.tester.assertTextPresent("GTG Data Feed");
 }
}
```

リスト 3 では、ログインを検証するテストと、アカウント情報を検証するテストの 2 つが定義されています。verifyAccountInfo テストでは、Test アノテーションの dependsOnMethods = {"verifyLogIn"} 節を使用して、このテストが verifyLogIn() メソッドに依存していることを指定している点に注意してください。

このテストを例えば TestNG の Eclipse プラグインを使用して実行した場合、verifyLogIn テストが失敗しても、図 1 に示すように TestNG は verifyAccountInfo テストをスキップするだけとなります。

図 1. TestNG でスキップされたテスト



失敗にするのではなくスキップするという TestNG の芸当は、大規模なテスト・スイートでの負担をまさに取り除いてくれます。チームはテスト・スイートの 50 パーセントが失敗した理由を突き止める代わりに、50 パーセントがスキップされた理由に集中して取り組めます。さらに嬉しいことに、TestNG では、失敗したテストだけを再実行するメカニズムによって依存関係のテストのセットアップを完全なものにします。

失敗した場合は再実行する

失敗したテストを再実行できる機能は特に大規模なテスト・スイートで役立ちますが、これは TestNG にしかない機能です。JUnit 4 では、例えばテスト・スイートが 1000 のテストで構成されている場合、そのうちの 3 つのテストが失敗すると、スイート全体を (修正して) 再実行しなければならないことになります。言うまでもなく、このような作業は何時間もかかります。

TestNG では、失敗が発生すると常に、失敗したテストを区別する XML 構成ファイルが作成されます。このファイルで TestNG ランナーを実行すると、TestNG は失敗したテストのみを実行します。つまり、前述の例では、スイート全体ではなく 3 つの失敗したテストを再実行すればいいだけのことになります。

リスト 2 の Web テストを使って、自分の目で確かめてみてください。verifyLogIn() メソッドが失敗すると、TestNG が自動的に testng-failed.xml ファイルを作成します。このファイルは、リスト 4 の代替テスト・スイートとして使用できます。

リスト 4. 失敗したテストの XML ファイル

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite thread-count="5" verbose="1" name="Failed suite [HistoryTesting]"
       parallel="false" annotations="JDK5">
 <test name="test.com.acme.ceg.AccountHistoryTest(failed)" junit="false">
  <classes>
   <class name="test.com.acme.ceg.AccountHistoryTest">
    <methods>
     <include name="verifyLogIn"/>
    </methods>
   </class>
  </classes>
 </test>
</suite>
```

小規模なテスト・スイートを実行しているときには、この機能はそれほど大したことではないように思えますが、テスト・スイートのサイズが大きくなれば、その有難みがすぐに分かります。

パラメーターを使ったテスト

TestNG にはあって JUnit 4 にはないもう一つの興味深い機能は、パラメーターを使ったテストです。JUnit では、テスト用メソッドのパラメーター・グループをさまざまに変えようと思ったら、固有のグループごとにテスト・ケースを作成しなければなりません。大抵の場合、これはそれほどやっかいな作業ではありませんが、時にはビジネス・ロジックに非常に各種多彩なテストが必要になるシナリオに出くわすこともあります。

そのような場合、JUnit テスターは FIT などのフレームワークに切り替えて、表データでテストを制御できるようにすることがよくあります。これと同様の方法が、TestNG ではすぐに使用できる機能として用意されています。TestNG の XML 構成ファイルにパラメトリック・データを配置すると、単一のテスト・ケースを異なるデータ・セットを使って再使用し、さらには異なる結果まで得ることができます。これは、簡単なシナリオしか表明しないテストや、結合を効率的に検証しないテストを避けるには完璧な手法です。

リスト 5 に、Java 1.4 での TestNG テストを定義します。このテストは classname および size パラメーターを受け入れます。この 2 つのパラメーターによって、クラス階層が検証されます (つまり、java.util.Vector に渡すと、HierarchyBuilder が 2 の値を持つ Hierarchy をビルドします)。

リスト 5. TestNG のパラメーターを使ったテスト

```
package test.com.acme.da;

import com.acme.da.hierarchy.Hierarchy;
import com.acme.da.hierarchy.HierarchyBuilder;

public class HierarchyTest {
 /**
  * @testng.test
  * @testng.parameters value="class_name, size"
  */
 public void assertValues(String classname, int size) throws Exception{
  Hierarchy hier = HierarchyBuilder.buildHierarchy(classname);
  assert hier.getHierarchyClassNames().length == size: "didn't equal!";
 }
}
```

リスト 5 に示したのは、データを変えて何度も再使用できる汎用テストです。ここで、ちょっと考えてみてください。JUnit では、テストするパラメーターの組み合わせが 10 通りあるとすると、10 のテスト・ケースを作成しなければなりません。それぞれのテストは、テスト用メソッドに対するパラメーターが違うだけで、本質的には同じことを実行します。一方、パラメーターを使ったテストでは、1 つのテスト・ケースを定義して、目的のパラメーター・パターンを例えば TestNG のスイート・ファイルに組み込むことができます。これを行っているのが、リスト 6 です。

リスト 6. TestNG のパラメトリック・スイート・ファイル

```
<!DOCTYPE suite SYSTEM "http://beust.com/testng/testng-1.0.dtd">
<suite name="Deckt-10">
 <test name="Deckt-10-test">

  <parameter name="class_name" value="java.util.Vector"/>
  <parameter name="size" value="2"/>

  <classes>
   <class name="test.com.acme.da.HierarchyTest"/>
  </classes>
 </test>
</suite>
```

リスト 6 の TestNG スイート・ファイルは、テストに対して 1 つのパラメーターの組み合わせ (class_name は java.util.Vector で、size は 2) しか定義していませんが、可能性は無限です。これに付随する利点として、テスト・データを XML ファイルの非コード成果物に移せば、プログラマーでなくてもデータを指定することができます。

高度なパラメーターを使ったテスト

データ値を XML ファイルに組み込む方法は非常に役立つはずですが、テストには String やプリミティブ値では表すことができない複雑なパラメーターの型が必要になる場合があります。TestNG では、このようなシナリオを @DataProvider アノテーションで対処します。このアノテーションによって、複雑なパラメーターの型を簡単にテスト・メソッドにマップできます。例えばリスト 7 の verifyHierarchy テストでは、Class 型を使用する buildHierarchy メソッドをオーバーライドして、Hierarchy の getHierarchyClassNames() メソッドが適切な String 配列を戻すことを宣言しています。

リスト 7. TestNG での DataProvider の使用方法

```
package test.com.acme.da.ng;

import java.util.Vector;

import static org.testng.Assert.assertEquals;
import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;

import com.acme.da.hierarchy.Hierarchy;
import com.acme.da.hierarchy.HierarchyBuilder;

public class HierarchyTest {

 @DataProvider(name = "class-hierarchies")
```

```
 public Object[][] dataValues(){
  return new Object[][]{
   {Vector.class, new String[] {"java.util.AbstractList",
     "java.util.AbstractCollection"}},
   {String.class, new String[] {}}
  };
 }

 @Test(dataProvider = "class-hierarchies")
 public void verifyHierarchy(Class clzz, String[] names)
  throws Exception{
    Hierarchy hier = HierarchyBuilder.buildHierarchy(clzz);
    assertEquals(hier.getHierarchyClassNames(), names,
   "values were not equal");
 }
}
```

dataValues() メソッドは、verifyHierarchy テスト・メソッドのパラメーター値と一致するデータ値を多次元配列によって指定します。TestNG はこのデータ値を繰り返して verifyHierarchy を 2 回呼び出します。1 回目の呼び出しでは、Class パラメーターが Vector.class に設定され、String 配列パラメーターには "java.util.AbstractList" と "java.util.AbstractCollection" の 2 つの値が Strings として含まれます。素晴らしく便利だと思いませんか。

どちらか一方を選ぶ必要はありません

私にとって TestNG を差別化している機能について説明しましたが、JUnit でまだ使用できない機能は他にもあります。例えば、TestNG ではテスト・グループを使用して、ランタイムなどの機能に応じてテストをカテゴリー化できます。またリスト 5 でもわかるように、javadoc スタイルのアノテーションを使って Java 1.4 でも機能します。

この記事の冒頭で述べたように、JUnit 4 と TestNG は表面的には似ています。ただし、JUnit がコード単位に焦点を合わせているのに対して、TestNG は高位レベルのテストを対象としています。TestNG の柔軟性は特に大規模なテスト・スイートで役立ち、1 つのテストの失敗によって数千のテストからなるスイートを再実行しなければならないような事態には至りません。どちらのフレームワークにも独自の長所があるため、両方をうまく組み合わせて使用してはならない理由は何もありません。

著者について

Andrew Glover

Andrew Gloverは合衆国ワシントン特別区にある、Vanward TechnologiesのCTO（最高技術責任者）です。Vanward Technologiesは自動化テスト・フレームワークの構築を専門としており、ソフトウェアのバグ発生数や統合時間やテスト時間の減少、また全体的なコード安定性改善に貢献しています。