JUnit用アサーション・エクステンション

alphaWorksのパッケージを使うと、複合アサーションを持つユニット・テストが容易に

Tony Morris (tonymorr@au1.ibm.com)
Software Engineer
IBM

2005年 3月 08日

JUnitを使うと、目的とする要求に合致しているというアサーション（assertions）を行うことでソフトウェアのコード・ユニットをテストできるようになりますが、こうしたアサーションは初歩的な操作用に限られています。この記事では、IBMのソフトウェア技術者のTony MorrisがJUnit用のJUnitX（Assertion Extensions for JUnit）を導入して、その隙間を埋めます。JUnitXは、JUnitフレームワーク内で実行する一連の複合アサーションを提供します。alphaWorksからの、この新しいパッケージを使って、Java™ソフトウェアの信頼性と堅牢性を向上させる方法を学んでください。

人気のJUnit自動ユニットテスト・フレームワークを使うと、「そのコードは目的とする要求に合致している」というアサーションを行うことによって、ソフトウェアのコード・ユニットをテストできるようになります。ところがこうしたアサーションは、「2つの変数に対して等しいとアサートする」とか「参照変数はヌルではないとアサートする」などの基本的な操作に限定されています。基本的なJUnitアサーションは便利ですが、実世界のソフトウェアのユニットテスト・シナリオで必要となるような、多くの複雑なアサーション機能を表すものではありません。

JUnitフレームワーク用のエクステンション・パッケージであるJUnitX（Assertion Extensions for JUnit）は、alphaWorks（参考文献）からダウンロードすることができます。JUnitXは、多くの一般的な複合アサーションで要求される実装を提供しています。ですからアサーション用に複雑なJUnitテスト・ケースを書く代わりに、JUnitXメソッドを呼び、（なんら追加設定無しに）同じコンテキストからアサーションを行うことができるのです。JUnitXはまた、独自のJUnitセルフテスト・スイートを含めることによって、ドキュメント通りに機能することをアサートします。これによって、JUnitXがJUnitXドキュメンテーションに従ってアサートしていることが改めて確認できます。つまりユニット・テストがフェールした場合には、JUnitXテスト実装が誤ったフェールをアサートしているのではなく、ソフトウェアのコード・ユニットがフェールしたのだと分かるのです。

JUnitXが有用となる典型的なシナリオとして、`java.lang.Object`クラスの`equals(Object)`メソッドと`hashCode()`メソッドが開始するコントラクト（contracts）があるでしょう。皆さんが開発する

クラスの中で、こうしたコントラクトを守るように要求される場合がよくあります。コントラクトを守ったアサーションをJUnitを使って行おうとすると、複雑なユニットテスト・ケースを開発する必要が出てきますが、これはエラーにつながりがちです。対照的にJUnitXを使ってアサーションを行うと、クラスのインスタンスを返すファクトリー実装を作り、ユニットテスト・ケースからJUnitXの`assertEqualsContract`メソッドと`assertHashCodeContract`メソッドを呼ぶだけ、というように簡単になります。

JUnitXの初歩

もう一つのJUnitX

この記事で説明しているJUnitXパッケージは、Extreme Javaにある、同じ名前のプロジェクトとは無関係です。Extreme JavaのJUnitXでは、プライベート・クラスや保護クラス、メソッド、変数などのテストができるようになっています。

JUnitXを効果的に使うために学ぶべきことは、殆どありません。JUnit自動ユニットテスト・フレームワークを直接使う方法を知っている人であれば、JUnitXエクステンション・パッケージを使うのは簡単なはずです。最初のステップは下記の通りです。

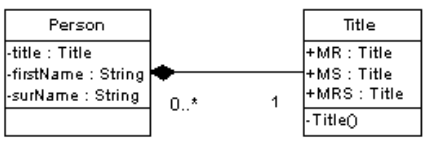
1. まず、JUnitテスト・ケースを実行するための環境を設定します。これには、「Automating the build and test process」（ 参考文献 ）に詳述されている手順に従った方が良いかも知れません。
2. JUnitXパッケージ（ 参考文献 ）をダウンロードし、JUnitXアーカイブを自分の好きなディレクトリーで解凍します。
3. lib/junitx.jarファイルが、JUnitテストを行うクラス・ローダーで使えるようにします。

こうすると、`junitx.framework.Assert`クラスに対してメソッド・コールをし、通常のJUnitテスト環境で`junit.framework.Assert`クラスを使うのと似た方法で機能をアサートすることができます。JUnitXのオンラインAPIドキュメンテーション（ 参考文献 ）には、`junitx.framework.Assert`クラスで使用できるメソッド・コールの詳細が説明されています。

ユース・ケース・シナリオ

皆さんが、ある人を表現するクラスを実装するように要求されたと思ってください。`Person`クラスには、その人の肩書き（ title ）、名前（ first name ）と名字（ surname ）、という3つの属性が必要です。肩書き属性として考えられる値は、MR、MS、MRSのいずれかなので、`Title`クラスをタイプセーフ列挙（ Typesafe Enumeration ）設計パターンを使って実装することにします。図1のUML図は、こうした要求事項の現状を示しています。

図1. 要求のUML図



リスト1は、こうした要求に対するソース・コードを示しています。

リスト1. 必要なクラスに対するソース・コード

```
public class Person {
  private Title title;
  private String firstName;
  private String surName;
}

public class Title {
  public static final Title MR = new Title();
  public static final Title MS = new Title();
  public static final Title MRS = new Title();
  // private constructor to prevent outside instantiation
  private Title() {
  }
}
```

ここでソース・コードは、堅牢かつ完全に機能するようになっている必要があります。そのためには、下記に示すような典型的な要求事項を含めて、より具体的な要求事項を満足しなければなりません。

- `Person`クラスは、コレクション・タイプ（Collection types）で効果的に使えるように、コントラクトに従って`equals(Object)`メソッドと`hashCode()`メソッドをオーバーライドする。
- `Person`クラスは`java.io.Serializable`インターフェースを実装し、なんらエラーを生じることなく、直列化と非直列化を行う。
- `Person`クラスは、サブクラス化できるように、`final`としては宣言しない。
- `Title`クラスは何のコンストラクターもエクスポーズしないため、`final`と宣言される（つまり`Title`クラスはサブクラス化できないことを意味します）。`final`修飾子がソース・コードや生成されたAPIドキュメンテーションの中で見えるように、この設計判断を理解していることを文書化しておくことのは良い習慣と言うことができます。
- `Title`クラスは`java.io.Serializable`インターフェースを実装し、なんらエラーを生じることなく直列化と非直列化を行い、また（タイプセーフ列挙設計パターンの要求に従って）同じインスタンスへと直列化する。
- `Title`クラスは、クラス外からのインストールを防ぐために、`private`デフォルト（つまり属性のない）コンストラクターを持つ。

こうした要求は全て、JUnitテスト環境でJUnitXを使うことによってアサートすることができます。リスト2のソース・コードは、「JUnitXの機能を使って全要求事項に合致できている」とアサートするJUnitテスト・ケースの一式です。

リスト2. JUnitXアサーションを使うJUnitテスト・ケース

```
import junit.framework.TestCase;
import junitx.framework.ObjectFactory;
import junitx.framework.Assert;
import java.io.Serializable;
import java.lang.reflect.Constructor;
public class TestRequirements extends TestCase {
  public void testPersonEqualsAndHashCodeContract() {
    // Different surnames should be unequal.
    ObjectFactory factory = new ObjectFactory() {
      public Object createInstanceX() {
        return new Person(Title.MR, "Bob", "Brown");
      }
```

```
      public Object createInstanceY() {
        return new Person(Title.MR, "Bob", "Smith");
      }
    }
    // Make sure the object factory meets its contract for testing.
    // This contract is specified in the API documentation.
    Assert.assertObjectFactoryContract(factory);
    // Assert equals(Object) contract.
    Assert.assertEqualsContract(factory);
    // Assert hashCode() contract.
    Assert.assertHashCodeContract(factory);
  }
  public void testPersonSerialization() {
    // Assert that the Person class directly implements Serializable.
    Assert.assertDirectInterfaceOf(Person.class, Serializable.class);
    // Assert that the Person instance can be serialized and deserialized without errors.
    Assert.assertSerializes(new Person(Title.MR, "Joe", "Blog"));
  }
  public void testPersonNotFinal() {
    // Assert that the Person class is not declared final.
    Assert.assertNotFinal(Person.class);
  }
  public void testTitleFinal() {
    // Assert that the Title class is declared final.
    Assert.assertFinal(Title.class);
  }
  public void testTitleSerialization() {
    // Assert that the Title class directly implements Serializable.
    Assert.assertDirectInterfaceOf(Person.class, Serializable.class);
    // Assert that the Title instances can be serialized and deserialized without errors.
    Assert.assertSerializes(Title.MR);
    Assert.assertSerializes(Title.MS);
    Assert.assertSerializes(Title.MRS);
    // Assert that serialization results in the same instance.
    Assert.assertSerializesSame(Title.MR);
    Assert.assertSerializesSame(Title.MS);
    Assert.assertSerializesSame(Title.MRS);
  }
  public void testTitleConstructor() {
    // Assert that the Title class has a default constructor.
    Assert.assertClassHasConstructor(Title.class, null);
    try {
      // Get the default constructor.
      Constructor con = Title.class.getDeclaredConstructor(null);
      // Assert that the default constructor is declared private.
      Assert.assertPrivate(con);
    }
    catch(NoSuchMethodException nsme) {
      // Should never get here, even when test fails.
      throw new IllegalStateException();
    }
  }
}
```

リスト1の`Person`クラスと`Title`クラスは、全ての追加要求事項は満足しないため、リスト2のテスト・ケースはパスしません。今度は、ユニット・テスト・ケースがパスする（要求事項合致を示します）ように、新しい要求事項に合致するクラスを開発します。リスト3は、規定された要求事項に合致する実装の例を示しています。

リスト3. 追加の要求事項に合致するように改訂されたクラス

```
import java.io.Serializable;
public class Person implements Serializable {
  private Title title;
```

```
  private String firstName;
  private String surname;
  public Person(Title title, String firstName, String surname) {
    this.title = title;
    this.firstName = firstName;
    this.surname = surname;
  }
  public boolean equals(Object o) {
    // Performance optimization only.
    if(this == o) {
      return true;
    }
    if(o == null) {
      return false;
    }
    if(!(o instanceof Person)) {
      return false;
    }
    Person p = (Person)o;
    return title == p.title & firstName.equals(p.firstName) & surname.equals(p.surname);
  }
  public int hashCode() {
    final int oddPrime = 461;
    int result = 73;
    result = result * oddPrime + title.hashCode();
    result = result * oddPrime + firstName.hashCode();
    result = result * oddPrime + surname.hashCode();
    return result;
  }
}

import java.io.Serializable;
public final class Title implements Serializable {
  public static final Title MR = new Title();
  public static final Title MS = new Title();
  public static final Title MRS = new Title();
  private static int nextIndex = 0;
  private final int index = nextIndex++;
  private static final Title[] VALUES = new Title[]{MR, MS, MRS};
  private Title() {
  }
  // Ensure that the same instance is returned when deserialized.
  Object readResolve() {
    return VALUES[index];
  }
}
```

リスト3のクラスでJUnitテスト・ケースを実行すると、テストをパスします。これによって、与えられた要求事項にコードが合致していると結論することができます。

まとめ

JUnitXを使わずに、コードに対するユース・ケース・シナリオでアサートを行おうとすると、（JUnitXを使うよりも）ずっと大量の作業が必要となります。テスト・ケースで失敗した場合には、不正なソフトウェア・コード・ユニットではなく、不正なテスト・ケースが構成された事を意味し、診断のために、さらに作業が必要となります。一方JUnitXテスト・ケースがフェールし、ソフトウェア・コード・ユニット中の欠陥がすぐに明確に分からない場合には、一緒に含まれているセルフテスト・スイートのソース・コードを読んで、テスト・ケースをパスするコード・ユニットはどう見えるのかを調べればよいのです。

より多くの機能を要求されるのに合わせて、JUnitXパッケージは今後も進化を続けます。

著者について

Tony Morris

Tony Morrisは、オーストラリアのゴールド・コーストにあるIBM Tivoli's Labのソフトウェア技術者です。IBM Tivoli Privacy Managerに関する作業に従事しており、様々なサードパーティー・アプリケーション用のプライバシー・モニターを開発しています。Javaプラットフォームでのプログラミング経験が7年間あり、オーストラリアのゴールド・コーストのSchool of Information Technology at Griffith Universityで非常勤の講師をしています。Griffith Universityにて情報技術の学位を取得しており、Sunの認定プログラマーであり、またSun認定のJava 2 Platform開発者でもあります。