コード品質を追求する: カバレッジ・レポートに騙されないために

テスト・カバレッジを測ることで迷路に入り込んでいませんか

Andrew Glover
CTO
Stelligent Incorporated

2006年 1月 31日

テスト・カバレッジ・ツールを使うとユニット・テストに深みが増しますが、このツールは多くの場合、誤って使われています。今回はAndrew Gloverが、この領域での彼の長い経験を生かして、新しいシリーズ、『コード品質を追求する』を開始します。第1回目である今回は、カバレッジ・レポートに表れる数字が実際に何を意味するのか、また逆に数字に意味がない場合について詳しく見て行きます。そして、カバレッジ・ツールを初期段階で頻繁に使用してコード品質を向上するための、3つの原則を提案します。

皆さんは、大部分の開発者がコード品質を気にするようになる以前のことを覚えていますか。当時は、巧妙に配置されたmain() メソッドはアジャイルであり、またテスト用に充分とも思われていました。その頃から、だいぶ時間が経ちました。私は、今や自動テストが品質重視のコード開発における必須の一面となっていることは素晴らしいと思っています。ただし私が素晴らしいと思っているのは、それだけではありません。今やJava™開発者には、コード・メトリックスや静的分析その他、コード品質を測るためのツールが山のようにあります。さらに、リファクタリングさえも、一連のパターンの中にカテゴリー分けできるのです。

コード品質を保証する

コード品質に関する質問に答えを求めている人は、Code Qualityディスカッション・フォーラムを訪れてみてください。このフォーラムは、Andrew Gloverが調整役を務めています。

こうした新しいツールによって、コード品質を確保することが以前よりもずっと容易になりました。しかし私達は、こうしたツールの使い方を知っている必要があります。このシリーズでは、コード品質確保のための詳細に焦点を当てることにします。こうした詳細は、必ずしも明快ではない場合が多いようです。また、様々なツールや手法を紹介する他、次のような点についても説明するつもりです。

- 最も品質に影響を与えるコード部分の定義方法と、そうした部分の計測方法
- 品質保証の目標を設定するための方法と、それに従って開発計画を立てるための方法
- 自分の必要に合ったコード品質ツールや手法を決定するための方法

- コード品質確保を、開発作業の一部として『初期段階で、頻繁に』苦労なく効果的に行えるようにするための、ベスト・プラクティスの実装方法（そしてお粗末なプラクティスを排除するための方法）

今回はまず、最も一般的であり、また開発者が品質確保のためのツールキットに簡単に追加できるものとして、テスト・カバレッジの測定について取り上げることにします。

偽の金塊にご注意

夜のうちにビルドが終わり、皆が冷水器のそばで談笑しています。開発者も、彼らの監督者も、幾つかのクラスが非常によくテストされており、テスト・カバレッジが90%台にも達していることを知ると、NFL（National Football League）よろしく肩をたたき合っています。チームのメンバーは、誰も自信満々です。欠陥が遠い昔の思い出となり、weakやinferiorの責任となるにつれ、「考えずにリファクタリングしてしまえ」という声が遠くまで聞こえるようになります。しかし、小さな声で、それに反論する声が聞こえてきます。

『紳士淑女の皆さん！ カバレッジ・レポートに騙されてはいけません！』

私を悪く取らないでください。テスト・カバレッジ・ツールを使うことは、何も悪いことではありません。ユニット・テストに追加すべきものとして、テスト・カバレッジ・ツールは偉大なものです。しかし重要なことは、それを手に入れた後で、どのように情報を組み立てるか、なのです。ところが一部の開発チームは、そこで最初の誤りをおかしてしまうのです。

カバレッジ率が高いということは、単に大量のコードが実行された、ということにすぎません。コードが『充分に』実行されたことにはならないのです。皆さんがコード品質に注目するのであれば、テスト・カバレッジ・ツールがどのように動作するのか、逆にうまく動作しないのはどういう場合かを、よく理解する必要があります。そうすれば、単にカバレッジ率が高いことに満足することなく、こうしたツールを利用して貴重な情報を入手できるはずです。

カバレッジの測定

一般的に言ってテスト・カバレッジ・ツールは、既存のユニット・テスト・プロセスの中に容易に追加することができ、結果も手軽に利用できます。入手可能なツールの1つを単純にダウンロードし、AntやMavenのビルド・スクリプトを少し修正すれば、皆さんの同僚が冷水器のそばで話題にできるような、『テスト・カバレッジ・レポート』という新種のレポートが手に入るのです。fooやbarなどのパッケージが、非常に『高い』カバレッジを示すのを見ると心が安らぎます。そして、少なくともコードの一部は「バグ無し」だと保証されていると信じてしまい、安心したくなるものです。しかし、それは大きな間違いなのです。

カバレッジの測定方法には様々なタイプがありますが、大部分のツールは、『ライン・カバレッジ』（『ステートメント・カバレッジ』とも言われます）に焦点を当てています。また、一部のツールは『分岐カバレッジ』もレポートします。テスト・カバレッジの測定は、テスト・ハーネスを使ってコード・ベースを実行し、データ（テスト・プロセスが、そのライフタイム中に「触ったことのある」コードに対応するデータ）をキャプチャーすることによって行われます。その後で、こうしたデータが合成され、カバレッジ・レポートが作成されます。Javaの場合では、テスト・ハーネスは普通JUnitであり、一般的なカバレッジ・ツールとしては、CoberturaやEmma、Cloverなど、幾つかがあります。

『ライン・カバレッジ』は単純に、ある特定なコード・ラインが実行されたことを示します。あるメソッドの長さが10ラインであり、そのうちの8ラインがテスト実行で実行されたとすると、そのメソッドのライン・カバレッジは80%ということになります。このプロセスは、アグリゲート・レベルでも行われます。あるクラスが100ラインから成り、そのうち45ラインが実行されたとすると、このクラスのライン・カバレッジは45%です。同様に、コード・ベースが、コメントではない10,000ラインから成り、ある特定なテスト実行で3,500ラインが実行されたとすると、そのコード・ベースのライン・カバレッジは35%ということになります。

『分岐カバレッジ』をレポートするツールは、判断ポイント（論理的なANDやORを含む条件ブロックなど）のカバレッジを測ろうとします。ライン・カバレッジの場合と同じく、あるメソッドに2つの分岐があり、テストで両方がカバーされたとすると、そのメソッドの分岐カバレッジは100%である、と言うことができます。

問題は、こうした測定が、どの程度有用なのか、という点です。つまり、こうした情報は容易に入手できますが、それをどのように組み立てるべきかの判断は皆さん次第なのです。この先の幾つかの例を見ると、私の言う要点が明確になるでしょう。

コード・カバレッジの実際

リスト1は、クラス階層構造の概念を具体化するための単純なクラスです。あるクラスは、一連のスーパークラスを持つことができます。例えばVectorの親はAbstractList、その親はAbstractCollection、その親はObject、などです。

リスト1. クラス階層構造を表現するクラス

```
package com.vanward.adana.hierarchy;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class Hierarchy {
  private Collection classes;
  private Class baseClass;

  public Hierarchy() {
    super();
    this.classes = new ArrayList();
  }

  public void addClass(final Class clzz){
    this.classes.add(clzz);
  }
  /**
   * @return an array of class names as Strings
   */
  public String[] getHierarchyClassNames(){
    final String[] names = new String[this.classes.size()];
    int x = 0;
    for(Iterator iter = this.classes.iterator(); iter.hasNext();){
       Class clzz = (Class)iter.next();
       names[x++] = clzz.getName();
    }
    return names;
  }

  public Class getBaseClass() {
    return baseClass;
```

```
  }

  public void setBaseClass(final Class baseClass) {
    this.baseClass = baseClass;
  }
}
```

リスト1を見ると分かるように、Hierarchyクラスは、baseClassインスタンスと、そのスーパークラスの集合を持っています。リスト2のHierarchyBuilderは、buildHierarchyという2つの多重定義staticメソッドを通して、Hierarchyクラスを作ります。

リスト2. クラス階層構造ビルダー

```
package com.vanward.adana.hierarchy;

public class HierarchyBuilder {

  private HierarchyBuilder() {
    super();
  }

  public static Hierarchy buildHierarchy(final String clzzName)
    throws ClassNotFoundException{
      final Class clzz = Class.forName(clzzName, false,
          HierarchyBuilder.class.getClassLoader());
      return buildHierarchy(clzz);
  }

  public static Hierarchy buildHierarchy(Class clzz){
    if(clzz == null){
      throw new RuntimeException("Class parameter can not be null");
    }

    final Hierarchy hier = new Hierarchy();
    hier.setBaseClass(clzz);

    final Class superclass = clzz.getSuperclass();

    if(superclass !=
      null && superclass.getName().equals("java.lang.Object")){
       return hier;
    }else{
       while((clzz.getSuperclass() != null) &&
          (!clzz.getSuperclass().getName().equals("java.lang.Object"))){
             clzz = clzz.getSuperclass();
             hier.addClass(clzz);
       }
       return hier;
    }
  }
}
```

テストの時間です！

テスト・カバレッジに関する記事に、テスト・ケースが無いはずがありません。リスト3では、単純な、楽観的シナリオとして、3つのテスト・ケースを持つJUnitテスト・クラスを定義しています。これらのテスト・ケースは、HierarchyクラスとHierarchyBuilderクラスの両方を実行しようとしています。

リスト3. あのHierarchyBuilderをテストせよ！

```
package test.com.vanward.adana.hierarchy;
```

```
import com.vanward.adana.hierarchy.Hierarchy;
import com.vanward.adana.hierarchy.HierarchyBuilder;
import junit.framework.TestCase;

public class HierarchyBuilderTest extends TestCase {

  public void testBuildHierarchyValueNotNull() {
     Hierarchy hier = HierarchyBuilder.buildHierarchy(HierarchyBuilderTest.class);
     assertNotNull("object was null", hier);
  }

  public void testBuildHierarchyName() {
     Hierarchy hier = HierarchyBuilder.buildHierarchy(HierarchyBuilderTest.class);
     assertEquals("should be junit.framework.Assert",
       "junit.framework.Assert",
         hier.getHierarchyClassNames()[1]);
  }

  public void testBuildHierarchyNameAgain() {
     Hierarchy hier = HierarchyBuilder.buildHierarchy(HierarchyBuilderTest.class);
     assertEquals("should be junit.framework.TestCase",
       "junit.framework.TestCase",
         hier.getHierarchyClassNames()[0]);
  }

}
```

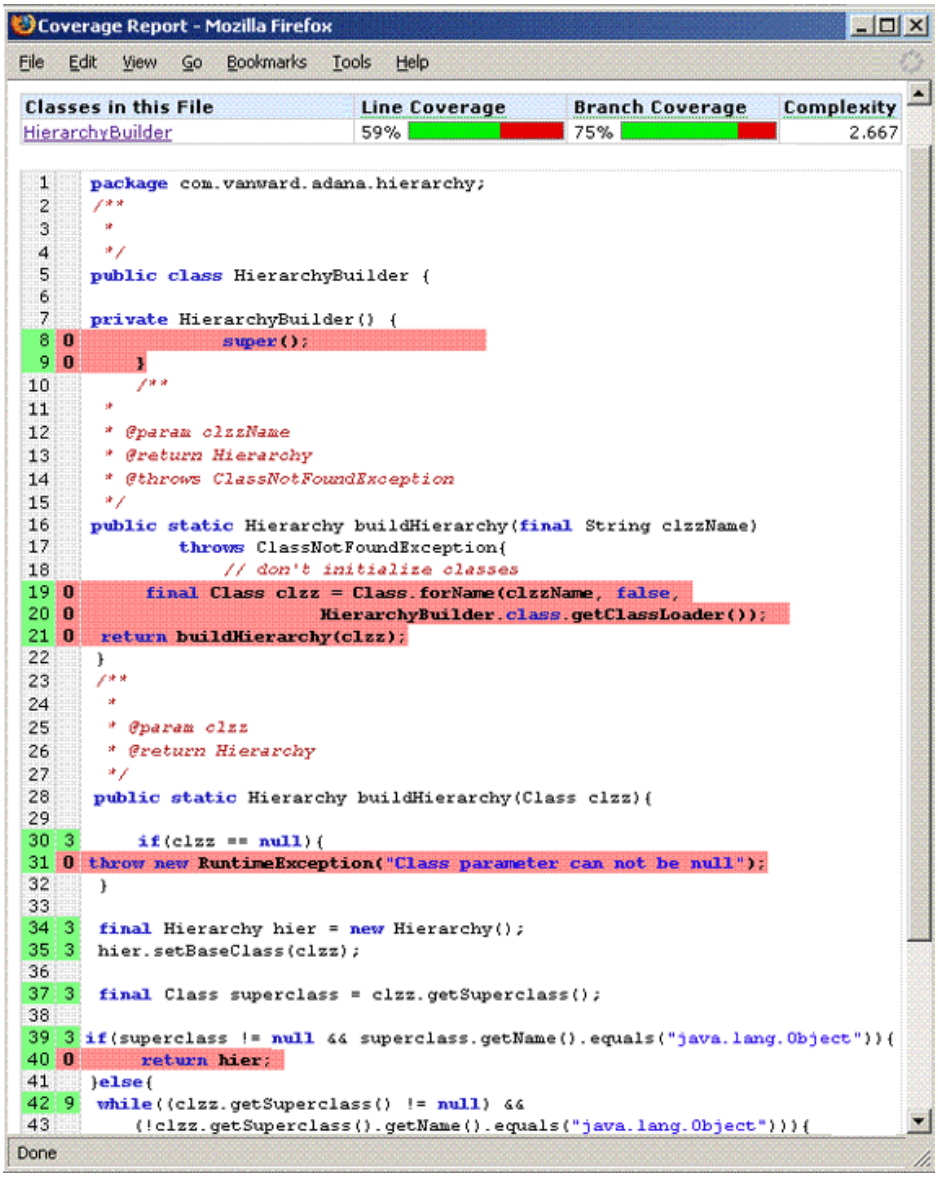
私は非常に貪欲なテスターなので、当然ながら少しばかりカバレッジ・テストを実行したいと思っています。Java開発者に入手可能なコード・カバレッジ・ツールの中で、私が好んで使っているのはCoberturaですが、これはレポートが分かりやすいからです。また、Coberturaはオープンソース・プロジェクトであり、先駆的なJCoverageプロジェクトから分かれたものです。

Coberturaのレポート

Coberturaのようなツールの実行は、JUnitテストを実行するのと同じくらい簡単です。単に、カバレッジをレポートするための特別なロジックを持ったテスト対象コードを実装する、という中間ステップがあるのみです（これはすべて、このツールのAntタスクやMavenのゴールによって処理されます）。

図1から分かるように、HierarchyBuilderに対するカバレッジ・レポートは、コードの中で『実行されなかった』幾つかのセクションを示しています。実際Coberturaは、HierarchyBuilderのライン・カバレッジが59%であり、分岐カバレッジが75%であると言っています。

図1. Coberturaのレポート



つまり、私が行ったカバレッジ・テストの第1試行では、幾つかのことがテストできていません。まず、パラメーターとしてStringをとる方のbuildHierarchy() メソッドが、全くテストされていません。第2に、もう一方のbuildHierarchy() メソッドの中にある2つの条件も実行されていません。面白いことに、懸念されるのは、実行されていない2番目のifブロックの方なのです。

この時点では、私は大して心配していません。単に幾つかのテスト・ケースを追加するだけのことだからです。気になる部分に到達できさえすれば、ひと安心というわけです。この、私の論理に注意してください。私は、カバレッジ・レポートを使って、テスト『されていない』ものを理解しようとしています。そうすることによって、このデータを使ってテストを機能強化するか、あるいは先に進むかという選択肢ができるのです。この場合では、幾つかの重要領域がカバーされずに残っているので、テストを機能強化することにします。

Coberturaの第2ラウンド

リスト4はJUnitテスト・ケースのアップデート版です。HierarchyBuilderを完全に実行するために、幾つかのテスト・ケースが追加されています。

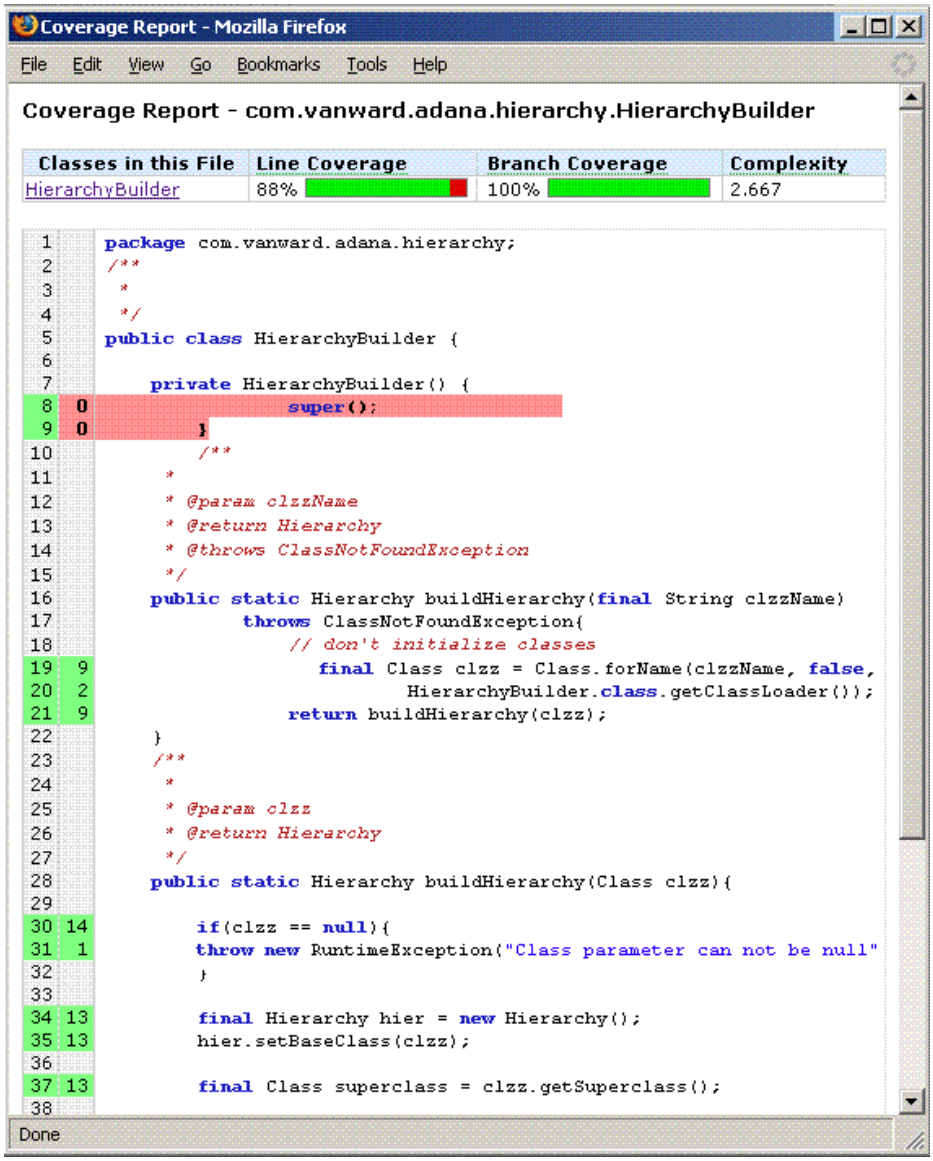
リスト4. JUnitテスト・ケースのアップデート版

```
package test.com.vanward.adana.hierarchy;

import com.vanward.adana.hierarchy.Hierarchy;
import com.vanward.adana.hierarchy.HierarchyBuilder;
import junit.framework.TestCase;

public class HierarchyBuilderTest extends TestCase {

  public void testBuildHierarchyValueNotNull() {
     Hierarchy hier = HierarchyBuilder.buildHierarchy(HierarchyBuilderTest.class);
     assertNotNull("object was null", hier);
  }

  public void testBuildHierarchyName() {
     Hierarchy hier = HierarchyBuilder.buildHierarchy(HierarchyBuilderTest.class);
     assertEquals("should be junit.framework.Assert",
       "junit.framework.Assert",
         hier.getHierarchyClassNames()[1]);
  }

  public void testBuildHierarchyNameAgain() {
     Hierarchy hier = HierarchyBuilder.buildHierarchy(HierarchyBuilderTest.class);
     assertEquals("should be junit.framework.TestCase",
       "junit.framework.TestCase",
         hier.getHierarchyClassNames()[0]);
  }

  public void testBuildHierarchySize() {
     Hierarchy hier = HierarchyBuilder.buildHierarchy(HierarchyBuilderTest.class);
     assertEquals("should be 2", 2, hier.getHierarchyClassNames().length);
  }

  public void testBuildHierarchyStrNotNull() throws Exception{
    Hierarchy hier =
       HierarchyBuilder.
       buildHierarchy("test.com.vanward.adana.hierarchy.HierarchyBuilderTest");
    assertNotNull("object was null", hier);
  }

  public void testBuildHierarchyStrName() throws Exception{
    Hierarchy hier =
       HierarchyBuilder.
       buildHierarchy("test.com.vanward.adana.hierarchy.HierarchyBuilderTest");
    assertEquals("should be junit.framework.Assert",
      "junit.framework.Assert",
        hier.getHierarchyClassNames()[1]);
  }

  public void testBuildHierarchyStrNameAgain() throws Exception{
    Hierarchy hier =
       HierarchyBuilder.
       buildHierarchy("test.com.vanward.adana.hierarchy.HierarchyBuilderTest");
    assertEquals("should be junit.framework.TestCase",
      "junit.framework.TestCase",
        hier.getHierarchyClassNames()[0]);
  }

  public void testBuildHierarchyStrSize() throws Exception{
     Hierarchy hier =
```

```
        HierarchyBuilder.
        buildHierarchy("test.com.vanward.adana.hierarchy.HierarchyBuilderTest");
     assertEquals("should be 2", 2, hier.getHierarchyClassNames().length);
  }

  public void testBuildHierarchyWithNull() {
     try{
       Class clzz = null;
       HierarchyBuilder.buildHierarchy(clzz);
       fail("RuntimeException not thrown");
     }catch(RuntimeException e){}
  }
}
```

この新しいテスト・ケースを使ってテスト・カバレッジ・プロセスを再度実行すると、より完全なレポートを得ることができます（図2）。これで、テストされていなかったbuildHierarchy() メソッドをカバーすることができ、もう一方のbuildHierarchy() メソッドの中にある両方のifブロックもテストできています。しかしHierarchyBuilderのコンストラクターはprivateなので、それを私のテスト・クラスでテストすることはできません（また、テストできないことを気にもしません）。従って、私のライン・カバレッジは相変わらず88%というレベルにあります。

図2. 再度の機会は無いと言ったのは誰でしょう？



ご覧の通り、コード・カバレッジ・ツールを使うことによって、対応するテスト・ケースを持たない重要コードを見つけ出すことが可能なのです。重要なことは、レポートを見る場合には（特に、高い値を示している場合には）、そうしたレポートの影にタチの悪い微妙な問題が隠れている場合もあるため、よく注意する、ということです。では次に、高いカバレッジ率の背後に隠れがちなコードの問題の例を、もう少し見てみましょう。

条件分岐に関する問題

ここまでで、コードの中にある多くの変数には1つ以上の状態があることが、よく分かったと思います。さらに、条件分岐があることによって、複数の実行パスが作られます。こうした注意点を念頭に置いて、1つのメソッドのみを持つ、バカバカしいほど単純なクラスを作ってみました（リスト5）。

リスト5. 下記の問題点が分かりますか？

```
package com.vanward.coverage.example01;

public class PathCoverage {

  public String pathExample(boolean condition){
    String value = null;
    if(condition){
      value = " " + condition + " ";
    }
    return value.trim();
  }
}
```

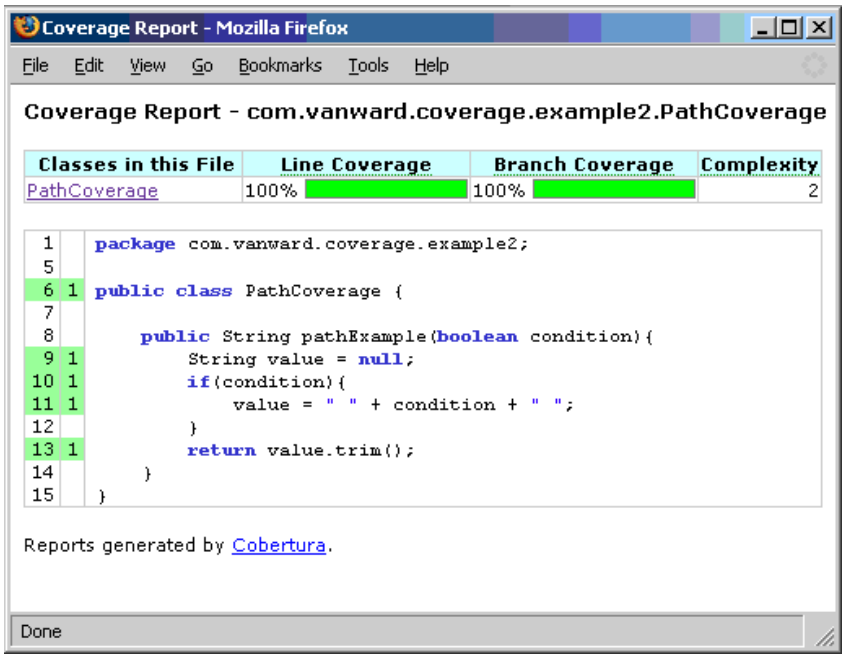
リスト5には、知らぬ間に進行する欠陥があることに気が付いたでしょうか。しかし気が付かなくても心配する必要はありません。リスト6は、pathExample() メソッドを実行し、それが正しく動作することを確認するテスト・ケースです。

リスト6. JUnitが救いに！

```
package test.com.vanward.coverage.example01;

import junit.framework.TestCase;
import com.vanward.coverage.example01.PathCoverage;

public class PathCoverageTest extends TestCase {

  public final void testPathExample() {
    PathCoverage clzzUnderTst = new PathCoverage();
    String value = clzzUnderTst.pathExample(true);
    assertEquals("should be true", "true", value);
  }
}
```

私のテスト・ケースは問題なく実行され、そして私の便利なコード・カバレッジ・レポート（下記の図3）を見ると、テスト・カバレッジは100%であり、私はまるでスーパースターのようです。

図3. まるでロックのスターのようなカバレッジだ！



どうやら冷水器のそばで暇つぶしをする時間のようです。しかし、ちょっと待ってください。私は例のコードの欠陥を疑っていたのではないでしょうか。リスト5をよく見てみると、ライン13は実際のところ、もしconditionがfalseの場合にはNullPointerExceptionを投げてしまいます。はてさて、一体どうしたことでしょう？

ここから分かることは、テストが効果的かどうかを示すものとして、ライン・カバレッジはそれほど偉大ではない、ということです。

パスの恐怖

リスト7では、もう一つ、間接的ながら明らかな欠陥を持った単純な例を定義しています。branchIt() メソッドの中にあるif条件の後ろ半分に注意してください。（ HiddenObjectクラスはリスト8で定義されています。 ）

リスト7.　このコードはごく単純です

```
package com.vanward.coverage.example02;

import com.acme.someotherpackage.HiddenObject;

public class AnotherBranchCoverage {

  public void branchIt(int value){
    if((value > 100) || (HiddenObject.doWork() == 0)){
      this.dontDoIt();
    }else{
      this.doIt();
    }
  }

  private void dontDoIt(){
    //don't do something...
  }

  private void doIt(){
```

```
    //do something!
  }
}
```

そうです、リスト8のHiddenObjectが悪いのです。リスト7の場合と同じようにdoWork() メソッドを呼ぶと、RuntimeExceptionとなります。

リスト8. あら何と！

```
package com.acme.someotherpackage.HiddenObject;

public class HiddenObject {

  public static int doWork(){
    //return 1;
    throw new RuntimeException("surprise!");
  }
}
```

しかし、うまいテストによって、例外をキャプチャーできるのです。リスト9も、もう一度ロックスターになるための、楽観的なテストです。

リスト9. JUnitでリスクを回避する

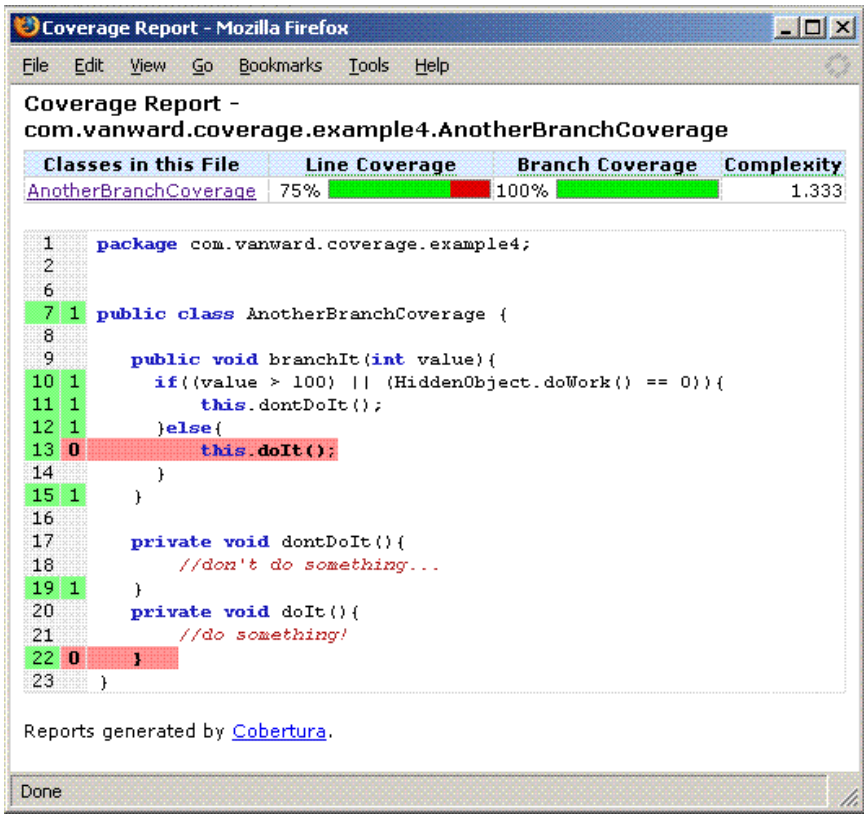
```
package test.com.vanward.coverage.example02;

import junit.framework.TestCase;
import com.vanward.coverage.example02.AnotherBranchCoverage;

public class AnotherBranchCoverageTest extends TestCase {

  public final void testBranchIt() {
    AnotherBranchCoverage clzzUnderTst = new AnotherBranchCoverage();
    clzzUnderTst.branchIt(101);
  }
}
```

皆さんは、このテスト・ケースについてどう思いますか？ 皆さんは私よりも少し多くテスト・ケースを書いたことがあるでしょうから、リスト7の例の怪しげな条件分岐に、1つ以上の短絡オペレーションがあると考えてみてください。最初の半分のロジックが、単純なint比較よりも少しばかり理知的だったと考えてみてください。どれだけのテスト・ケースを書けば皆さんは満足できるでしょう。

単純に数をください

ここまで来れば、リスト7や8、9に対するテスト・カバレッジ分析の結果は驚くにあたらないでしょう。図4のレポートを見ると、75%のライン・カバレッジと100%の分岐カバレッジが達成されています。最も重要なこととして、ライン10が実行されているのです！

図4. 空しい報酬



まあ、何と素晴らしい結果でしょう。少なくとも、ちょっと見る限りはその通りです。皆さんは、このレポートが誤解を生みやすいことが分かるでしょうか。雑に見てしまうと、コードは『充分にテストされた』と思えてしまう可能性があります。それに基づいて、欠陥による危険性は非常に低いと思われてしまうかも知れません。つまりこのレポートは、or短絡回路の後半が時限爆弾であることを見つける上では、ほとんど何の役にも立たないのです。

品質向上のためのテスト

もう一度、言っておきましょう。テスト・カバレッジ・ツールはテスト・プロセスの一部として使うことができ、また使うべきですが、『カバレッジ・レポートに騙されてはいけません』。カバレッジ・レポートは、適切にテストされていないコードを検出するために使うべきものである、ということを第一に理解しておく必要があります。カバレッジ・レポートを検証する場合には、低い値を探し出し、その特定なコードが、なぜ完全にテストできなかったのかを理解すべきなのです。そうしたことを知った上で、開発者や監督者、QA専門家は、テスト・カバレッジ・ツールが本当に役に立つところでカバレッジ・ツールを使えばよいのです。カバレッジ・ツールが役立つ場合として、次のような3つの一般的シナリオを挙げることができます。

- 既存コードを修正するための時間を見積もる
- コード品質を評価する
- ファンクショナル・テストを予測評価する

テスト・カバレッジによって迷路に入り込みがちな場合を幾つか示したので、次のようなベスト・プラクティスを活用することを考えてみましょう。

1. 既存コードを修正するための時間を見積もる

コードに対してテスト・ケースを書くことによって、開発チーム全体の自信が自然に高まります。テストされたコードは、対応するテスト・ケースを持たないコードよりもリファクタリングや維持管理、機能向上が容易です。テスト・ケースは、対象となるコードが『どのように』動作するかを暗黙的に示すため、よく練られたドキュメンテーションとして扱うこともできます。さらに、コメントやJavadocsなどの静的なコード・ドキュメンテーションとは異なり、対象のコードが変更されると、通常はテスト・ケースも並行して変更されるものです。

逆に言うと、対応するテストを持たないコードは理解しにくく、『安全に』変更することが、より困難です。つまり開発者や監督者は、コードがテストされたかどうかを知ることによって、またテスト・カバレッジの実際の数字を見ることによって、既存コードを修正するために必要な時間を、より正確に予測できるようになるのです。

冷水器の近くに行くと、私の言う要点がよく分かるでしょう。

> マーケティング部門のLinda: 「ユーザーが取引を実行した時に、システムがxをするようにしたい。それにはどの位時間がかかるでしょう。私達の顧客は、早急にこの機能を必要としています。」
> 監督者であるJeff: 「そうだな。あのコードは数ヶ月前にJoeが書いたものだ。あなたが言うような変更だと、ビジネス・レイヤーの変更と、幾つかのUI変更が必要になる。Maryが2、3日で変更を仕上げられるはずだ。」
> Linda: 「Joeって誰ですか？」
> Jeff: 「ああそうだった。彼は自分の仕事を理解しないのでクビにしたのだった。」

こうした状況は、ちょっと不気味だと思いませんか。それでも、Jeffは新しい機能についての作業をMaryに割り当て、Maryも、2日もあれば仕上がると思っています。いやつまり、コードを見るまでは、の話ですが。

> Mary: 「Joeは寝ながらこのコードを書いたのでしょうか？ これは私が今まで見た中で最悪のコードです。一体これがJavaコードなのかすら分かりません。これを完全に壊さない限り、変更などできません。辞めさせてもらいます。」

どうやら、冷水器のそばに集まったチームの具合はあまり良くないようです。では、この不幸な出来事のシリーズを巻き戻して、今度はJeffとMaryにカバレッジ・レポートで力を与えましょう。Lindaが新しい機能を要求した時にJeffが最初にすることは、以前のビルドに対して、例の便利なカバレッジ・レポートを調べることです。Jeffは、変更を要する特定パッケージのカバレッジがほとんどゼロであることに気が付き、Maryに相談します。

> Jeff: 「Joeが書いたコードはひどい。しかも、そのほとんどが、テストされていない。Lindaの要求をサポートするためには、どの位長くかかると思いますか。」
> Mary: 「あのコードはひどい。見る気にもなりません。Markではできませんか。」
> Jeff: 「たった今、テストを書かないのでMarkをクビにしたところだ。このコードをテストし、変更を加えることを、あなたに任せる。どの位時間が必要か言いなさい。」
> Mary: 「テストを書くために、少なくとも2日必要です。それからコードをリファクターし、その後で機能を追加します。合計で約4日くらいだと思っています。」

お分かりの通り、知識は力なのです。開発者は、コード修正を行う前に、カバレッジ・レポートを使ってコード品質をチェックできます。同様に監督者もカバレッジ・データを使うことによって、開発者が実際に作業のために必要とする時間を、より正確に見積もることができるのです。

2.コード品質を評価する

開発者がテストを行うことによって、コード中の欠陥を減少させることができるため、今や多くの開発チームでは、新しく開発するコードや修正すべきコードと同時にユニット・テストも書くことを要求するようになっています。しかし、上記のMarkの例でも分かるように、ユニット・テストは必ずしもコーディングと並行して行われるわけではないため、コード品質が低下する可能性があります。

開発チームはカバレッジ・レポートを監視することによって、対応するテスト無しに大きくなりつつあるコードを素早く見つけることができます。例えば、週の最初にカバレッジ・レポートを実行した結果、プロジェクトの中で鍵となるパッケージのカバレッジが70%だったとします。その週の後の方になって、そのパッケージのカバレッジが60%に落ちたとすると、その原因は下記のいずれかであると推論することができます。

- そのパッケージのライン数は増加しているにもかかわらず、新しいコードに対する対応テストが書かれていない（あるいは、新しく追加されたテストが、新しいコードを効果的にカバーしていない）
- テスト・ケースが除去されている
- 両者が同時に起きている

ここで素晴らしいことは、傾向を見られる、ということです。レポートを定期的に見ることによって、ゴールの設定（カバレッジ率を取得する、テスト・ケース対コード・ライン数の比を維持する、など）が容易になり、またその進み具合の監視も容易になります。もし、テストが書かれないことが定常的であることに気が付いた場合には、先回りしたアクションをとることができます（つまり開発者を訓練する、指導する、仲間とプログラミングさせる、など）。反応を予測できる方が、顧客が「致命的な欠陥」を見つける（そうした欠陥は単純なテストによって何ヶ月も前に検出できたはずです）よりも、あるいはユニット・テストが行われていないことを監督層が知って驚く（そして怒る）よりも、ずっと良いのです。

カバレッジ・レポートを使って適切なテストを確保することは、非常に良い習慣です。問題は、それを身に付いた習慣として行うようにすることです。そのためには、例えば『継続的統合』プロセスの一部として、カバレッジ・レポートを毎晩生成し、見るようにすることです。

3. ファンクショナル・テストを予測評価する

コード・セクションに対して適当なテストが『無いこと』を示す上で、コード・カバレッジ・レポートが最も効果的なことから、QAのメンバーはこのデータを使って、ファンクショナル・テストに関する懸念領域を予測評価することができます。冷水器に戻り、QAのリーダーであるDrewがJoeのコードについて何を言うかを聞いてみましょう。

> DrewがJeffに:「私達は、次のリリースのためのテスト・ケースを書き上げつつある。ところが、基本的にコード・カバレッジが全くない大量のコード領域に気が付いた。これは株取引に関するコードらしい。」

> Jeff: 「ああ、そうそう。その領域には少し問題があります。私が賭け事師なら、そのファンクション領域に特に注意するでしょう。そのアプリケーションの別の主要変更部分に関する作業をMaryが行っています。彼女はユニット・テストを適切に書いていますが、だからといってコードが完全なはずはありません。」
> Drew: 「そう。私は作業のためのリソースとレベルを決めようとしているところです。どうやら今回も手不足になりそうです。チームには株取引に集中するようにさせましょう。」

繰り返しますが、知識は力なのです。ソフトウェア・ライフサイクルにおける他の利害関係者（QAなど）と注意深く協力作業を行う中で、カバレッジ・レポートが提供する情報を活用すれば、リスクを緩和することができます。上のシナリオでは、JeffはDrewのチームに初期リリースを渡すことができます。この初期リリースは、Maryが行う変更のすべては含んではいないかも知れません。しかしDrewのチームは、これによってアプリケーションの中の株取引部分に集中することができます（どうやらこの部分は、対応するユニット・テストを持つコードよりも、欠陥によるリスクが高そうなのです）。

効果的なテストを行うために

テスト・カバレッジ・ツールは、ユニット・テストの仕組みの中にぜひ加えるべきものです。テスト・カバレッジを測ることによって、テストという既に有利なプロセスに、深みと精度が加わります。ただし、カバレッジ・レポートを見る場合には、よく注意する必要があります。単にカバレッジ率が高いからと言って、コード品質が保証されることにはなりません。カバレッジ率が高いことは、欠陥を含む可能性が「より低い」というだけであって、必ずしも欠陥が無いことを意味するわけではありません。

テスト・カバレッジ測定の秘訣は、カバレッジ・レポートを、「テストされていない」コードをマクロ・レベルとマイクロ・レベルの両面から検出する目的に使うことです。コード・ベースを最上位レベルから分析することによって、また個々のクラスのカバレッジを分析することによって、より深いカバレッジ・テストを行うことができます。この原則を採り入れさえすれば、プロジェクトに必要な時間の見積もりのために、あるいは継続的にコード品質を監視するために、そしてQAとの共同作業の効率改善のために、カバレッジ測定ツールを使えるようになるはずです。

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。