# Cobertura でテスト対象範囲を調べる

## バグが潜む未テストのコードを見つける

Elliotte Rusty Harold
Adjunct Professor
Polytechnic University

2005年 5月 03日

Cobertura は、テスト対象範囲を調べるオープンソースのツールであり、コードベースを実装することによって、またテスト・スイートが実行する際に、どのコード行が実行され、どのコード行が実行されないかを監視して、対象範囲を調べます。また、Cobertura は未テストのコードを特定し、バグを見つけることに加えて、到達不能な、死んでいるコードにフラグを立てることによって、コードを最適化します。その結果、API が現実的にどのように動作するかを洞察することもできます。この記事では Elliotte Rusty Harold が、コード・カバレッジ (テスト対象範囲) に関するベスト・プラクティスを使って Cobertura を活用する方法を解説します。

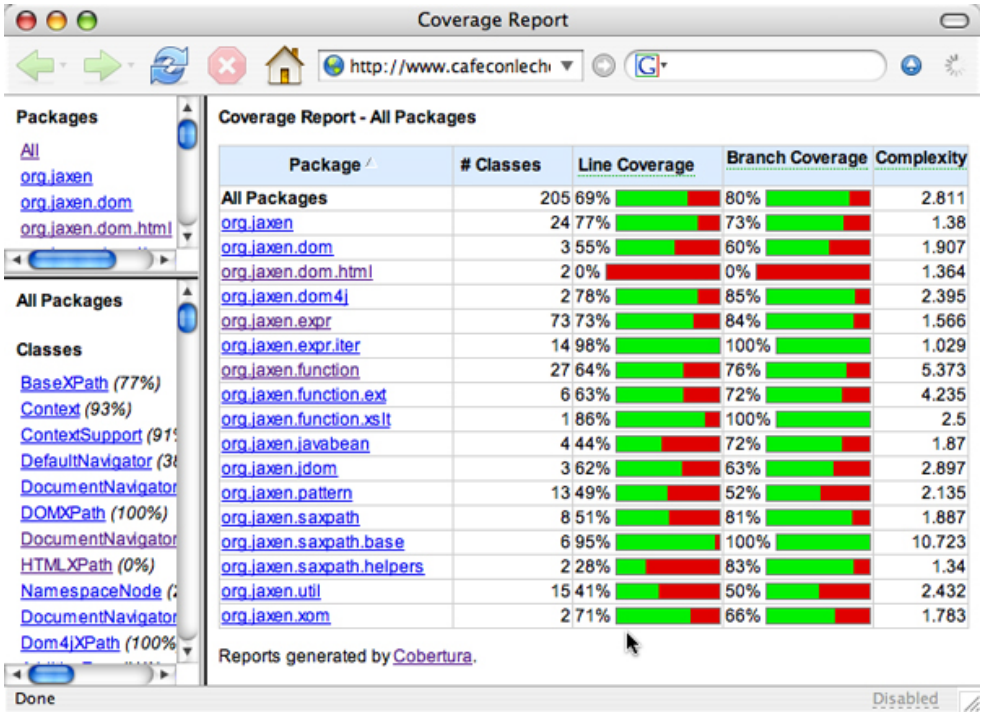
テスト駆動開発は、プログラミングに関する過去 10 年間の改革の中で、最も重要なものですが、「まずテスト」というプログラミング (test-first programming) やユニット・テストは、格別新しいものではありません。質の良いプログラマーであれば、こうした技法を半世紀も前から使っています。しかし、堅牢なバグ無しのソフトウェアを、スケジュール通り、予算通りに開発する上での致命的要素として認識されるようになったのは、ここ数年のことです。しかしテスト駆動開発は、テスト以上に良くはなりません。確かにテストによってコード品質は改善されますが、それは実際にテストされたコードベース部分について言えるだけです。プログラムのどの部分が未テストであるかを伝え、もっとバグが見つかるように、未テスト部分に対するテストが書けるようにするためのツールが必要なのです。

Mark Doliner による Cobertura は (cobertura はスペイン語で、coverage (対象範囲) を意味します) 、この問題に対応するための、フリーの GPL ツールです。Cobertura はログのための追加ステートメントを持ったバイトコードを実装してテストを監視し、テスト・スイートが実行する際に、どの行が実際にテストされ、どの行がテストされていないかを調べるのです。そして、具体的にどのパッケージ、クラス、メソッド、個々のコード行がテストされていないかを示す HTML または XML のレポートを作成します。ですから皆さんは残ったバグを見つけるために、こうした特定の領域に対してテストを書けばよいのです。

## Cobertura の出力を読む

まず、生成された Cobertura の出力から始めることにしましょう。図 1 は、Jaxen テスト・スイート (参考文献) で Cobertura を実行して得られたレポートです。カバレッジが、ほとんど全て (`org.jaxen.dom.html`パッケージは、ほぼ 100%) から、ほとんど無し (`org.jaxen.expr.iter`は全くカバーされていない) まで、大きな幅があることが分かります。

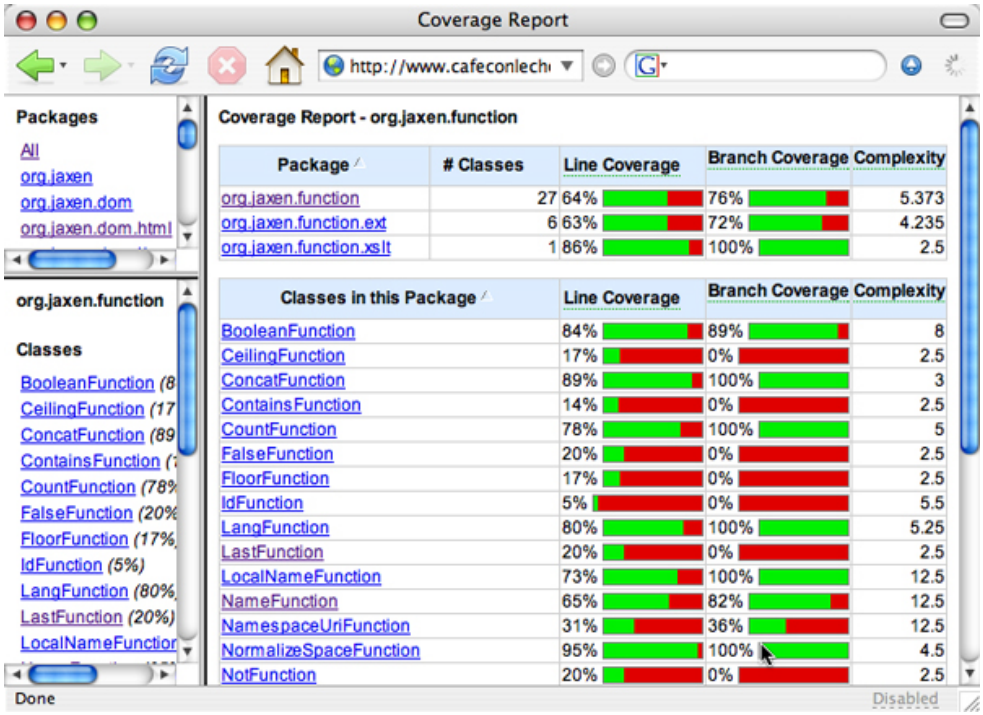
### 図 1. Jaxen に対する、パッケージ・レベルでのカバレッジの統計




Cobertura は、テストされた行数と、テストされた分岐数の両方によってカバレッジを計算します。最初のパスでは、この 2 つの違いは、あまり重要ではありません。Cobertura はまた、クラスに対する平均的な McCabe's cyclomatic complexity (繰り返し処理の複雑性、参考文献を参照) も計算します。

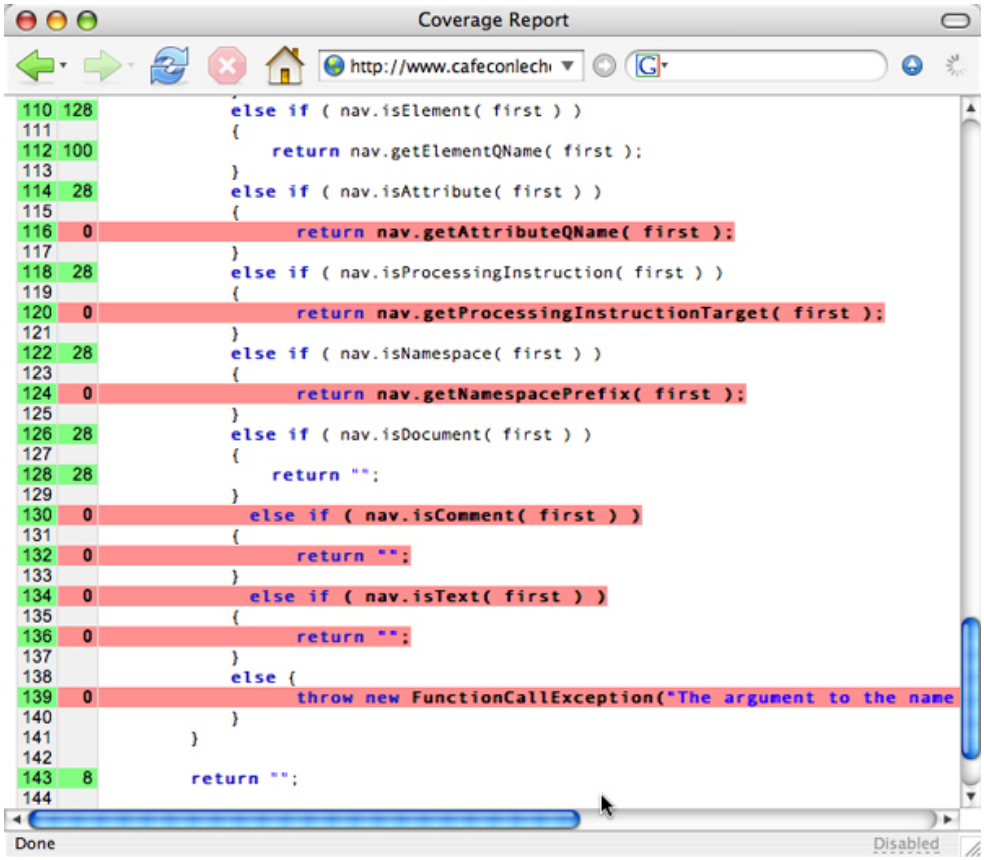
HTML レポートをさらにたどると、特定なパッケージ、またはクラスのカバレッジを見ることができます。図 2 は、`org.jaxen.function` パッケージに対するカバレッジ統計を示しています。このパッケージでのカバレッジは、`SumFunction` クラスに対する 100%から、`IdFunction` クラスに対する 5%までの幅があります。

図 2. org.jaxen.function パッケージでのコード・カバレッジ




さらに、各クラスにまでたどると、具体的にどの行がテストされていないかを見ることができます。図 3 は、`NameFunction` クラスでのカバレッジの一部を示しています。左側のカラムは行番号を示します。次のカラムは、テスト実行中に、その行が何度実行されたかを示します。見て分かる通り、112 行は 100 回実行され、114 行は 28 回実行されています。赤でハイライトされた行は、全くテストされていません。このレポートによって、メソッド全体としてはテストされているものの、多くの分岐がテストされていないことが分かります。

図 3.NameFunction クラスでのコード・カバレッジ




Cobertura は jcoverage から分かれたものです (参考文献) 。jcoverage の GPL 版は 1 年以上更新されておらず、その長年来のバグの幾つかを、Cobertura が修復しています。オリジナルの jcoverage 開発者達は、オープンソースで続ける代わりに、商用版の jcoverage と、同じコードベースを元にしたクローズドソース製品である jcoverage+とに注力しています。オリジナルの開発者が対価を要求することに決めたからといって、その成果まで死んでしまう必要はないというのは、正にオープンソースの驚異でしょう。

## 未テスト部分を識別する

Cobertura のレポートを使うと、コード中の未テスト部分を特定できるため、その部分に対するテストを書くことができます。例えば図 3 は、Jaxen がテキスト・ノードやコメント・ノード、命令処理ノード、属性ノード、名前空間ノードに対して、`name()` ファンクションを適用するテストを必要としていることを示しています。

カバーされていないコードが、Cobertura が特定するように大量にある場合には、足りないテスト全てを追加するのは非常に時間がかかります。しかしこれは、するだけの価値があるのです。全てを一度に行う必要はありません。まず、全くカバーされていないパッケージのような、一番テストされていないコードから始めるのです。全てのパッケージを少しテストしたら、カバーされていない各クラスに対するテストを書きます。全てのクラスを部分的にテストしたら、カバーされていないメソッドを対象にしたテストを書きます。全てのメソッドをテストしたら、未テストのステートメントをアクティブにするために、何が必要かを探し始めればよいのです。

## 未テストのコードを (ほとんど) 無くす

テストはできるが、テストすべきではない、というものがあるでしょうか。誰に質問するかによって、答えが異なるでしょう。J. B. Rainsberger は JUnit FAQ の中で、次のように書いています。「一般的な考え方は次の通りです。それ自体では壊れないのであれば、(簡単過ぎるので) 壊れることはありません。第一の例は `getX()` メソッドです。`getX()` メソッドが、インスタンス変数の値のみを答える、と考えてください。その場合 `getX()` は、コンパイラー、あるいはインタープリターが一緒に壊れない限り、壊れることはありません。ですから、`getX()` をテストすべきではありません。何の利点も無いのです。`setX()` についても同じことが言えますが、その `setX()` メソッドが、パラメーターに関して何らかの妥当性検証を行う場合や、何らかの副作用がある場合には、恐らくテストする必要があるでしょう。」

> 理論的には、カバーされていないコードに対するテストを書いても、バグが見つかる保証はありません。しかし私は現実問題として、そうしたテストでバグが見つからない、というケースを見たことがありません。未テストのコードにはバグが満ちています。テストが少なければ少ないほど、未発見のバグがコードの中に潜んでいる可能性が高いのです。

私は、彼の意見に同意できません。私が、「(簡単過ぎるので) 壊れることはない」コードの中で見つけたバグの数は数え切れません。確かに、一部のセッターやゲッターは、あまりに些細なため、フェールしようがありません。しかし私は、どのメソッドが (簡単過ぎるので) 壊れることはなく、どのメソッドがそう見えるだけなのか、判別できた試しがありません。セッターやゲッターのような単純なメソッドをカバーするテストを書くことは、決して難しくありません。そのために費やす時間は、予期せぬバグが見つかるメソッドの数だけで、十分に元が取れるはずです。

一般的に言って、いったん測定を始めれば、90%のテスト・カバレッジに達するのは、ごく簡単です。ところが 95%以上にまでカバレッジを増加させるには、ちょっとした細工が必要です。例えば、全バージョンのライブラリーには現れないバグに対する回避策をテストするために、別バージョンのサポート・ライブラリーをロードする方法があります。あるいはコードを再構成して、通常は到達できないコード部分に、テストが到達できるようにします。保護されたメソッドを単にパブリックにし、テストされるように、クラスを拡張する方法もあります。こうしたトリックは過剰に思えるかも知れませんが、私はこうした方法によって、未発見のバグを半分の時間で見つけることができています。

完璧な、100%のコード・カバレッジは、必ずしも達成できるものではありません。いかにコードをいじっても、何行か、あるいは幾つかのメソッド、あるいはクラス全体が、とにかくテストで到達できないことが、時にはあるのです。皆さんが突き当たりそうな壁の例としては、次のようなものでしょう。

- 特定なプラットフォームでのみ実行するコード。例えば、よく設計された GUI アプリケーションで「Exit」メニュー・アイテムを追加するコードは、Windows PC では実行しますが、Mac では実行しません。
- 起こり得ない例外を捉える `catch` 文。例えば `ByteArrayInputStream` から読み取る時に投げられる `IOException` など。
- 実際には呼び出されることはないものの、インターフェース契約を満足するために実装する必要のある、非パブリック・クラス内のメソッド。

- 例えば UTF-8 エンコーディングの認識失敗など、仮想マシンのバグを処理するコード・ブロック。

> ユニット・テスト・パッケージや、クラス自体も調べることを忘れないでください。私はこれまでに、テスト・メソッドやテスト・クラスが、テスト・スイートで実際には実行されないことに、一度ならず気がついたことがあります。通常はこれによって、名前付けでのバグ (例えば、あるメソッドに対して `testSomeReallyComplexCondition` の代わりに `tesSomeReallyComplexCondition` と名前を付ける) や、プライマリーの `suite()` メソッドに追加し忘れたクラスなどのバグで影響が出ます。またある場合には、予期せぬ条件が、テスト・メソッド内でコードがスキップされているためだったこともあります。いずれにせよ、私はテストを書いたのですが、実際には実行されていませんでした。JUnit は、(皆さんは実行していると思っている) 全テストは実行してはいない、と言ってくれませんが、Cobertura は言ってくれるのです。いったん見つかりさえすれば、修正は通常、簡単なものです。

これらの問題や、類似の問題があることを考えると、一部の極端なプログラマーが未テストのコード全てを自動的に削除してしまうのは、非現実的であり、滑稽でもあると思います。絶対的に完璧なコード・カバレッジが常に実現できないからといって、カバレッジを改善しなくて良いことにはなりません。

とはいえ、到達できないステートメントやメソッドは多くの場合、何ら目的を持たない、痕跡のようなコードと言うことができ、コードベースから削除しても何ら影響がありません。また、場合によっては、リフレクションを使ってプライベート・メンバーにアクセスする、非常に臭い手段を使って未テストのコードをテストすることも可能です。あるいは、テスト・クラスを、テストを行っているクラスと同じパッケージの中に置くことによって、未テストの、パッケージ保護されたコードに対するテストを書くこともできます。しかし、そんなことをすべきではありません。公開された (パブリックであり、保護された) インターフェースで到達できないコードは、そんなことをせずに削除すべきです。到達できないコードを、コードベースの一部にすべきではありません。コードベースは小さければ小さいほど、それを理解し、維持することは容易なのです。

## Cobertura を実行する

さて、コード・カバレッジを測定するメリットが分かったので、Cobertura を使って実際にコード・カバレッジを測定するための具体的な詳細を説明しましょう。Cobertura は Ant から実行するように作られています。まだ IDE プラグインは何もありませんが、1 年か 2 年のうちに、何かしら開発されるでしょう。

まず、build.xml ファイルにタスク定義を追加する必要があります。この、最上位レベルの `taskdef` 要素は、カレントの作業ディレクトリーに cobertura.jar ファイルがあることを規定します。

```
<taskdef classpath="cobertura.jar" resource="tasks.properties" />
```

次に、コンパイル済みのクラス・ファイルにログ用のコードを追加する `cobertura-instrument` タスクが必要です。`todir` 属性は、実装されたクラスをどこに置くべきかを指示します。`fileset` 子要素は、どの.class ファイルを実装すべきかを規定します。

```
<target name="instrument">
  <cobertura-instrument todir="target/instrumented-classes">
    <fileset dir="target/classes">
      <include name="**/*.class"/>
    </fileset>
  </cobertura-instrument>
</target>
```

通常、テスト・スイートを実行するのと同じタイプの Ant タスクとして、テストを実行します。違うのは、実装されたクラスが、オリジナル・クラスよりも前にクラスパスに現れる必要があること、Cobertura JAR ファイルをクラスパスに追加する必要があることだけです。

```
<target name="cover-test" depends="instrument">
  <mkdir dir="${testreportdir}" />
  <junit dir="./" failureproperty="test.failure" printSummary="yes"
         fork="true" haltonerror="true">
    <!-- Normally you can create this task by copying your existing JUnit
         target, changing its name, and adding these next two lines.
         You may need to change the locations to point to wherever
         you've put the cobertura.jar file and the instrumented classes. -->
    <classpath location="cobertura.jar"/>
    <classpath location="target/instrumented-classes"/>
    <classpath>
      <fileset dir="${libdir}">
        <include name="*.jar" />
      </fileset>
      <pathelement path="${testclassesdir}" />
      <pathelement path="${classesdir}" />
    </classpath>
    <batchtest todir="${testreportdir}">
      <fileset dir="src/java/test">
        <include name="**/*Test.java" />
        <include name="org/jaxen/javabean/*Test.java" />
      </fileset>
    </batchtest>
  </junit>
</target>>
```

Jaxen プロジェクトは、JUnit をテスト・フレームワークとして使用しますが、Cobertura はフレームワークを気にせず、TestNG でも Artima SuiteRunner でも HTTPUnit でも、あるいは皆さん自作のシステムでも、同じようにうまく動作します。

最後に、`cobertura-report` タスクが、この記事の最初の方で示したような HTML ファイルを生成します。

```
<target name="coverage-report" depends="cover-test">
 <cobertura-report srcdir="src/java/main" destdir="cobertura"/>
</target>
```

`srcdir` 属性は、オリジナルの.java ソースコード・ファイルがどこにあるかを規定します。`destdir` 属性は、Cobertura が出力 HTML を置くべきディレクトリー名を規定します。

同じようなタスクを皆さん独自の Ant ビルド・ファイルに追加すれば、下記のようにタイプするだけで、カバレッジ・レポートが生成されます。

```
% ant instrument
% ant cover-test
% ant coverage-report
```

もちろん、皆さんの好みに合わせて、ターゲットの名前を変更したり、これら 3 つのタスクを１つのターゲットにまとめたりすることもできます。

## まとめ

Cobertura は、アジャイルなプログラマーのツールボックスに追加すべき、重要なツールです。Cobertura を使うと、コード・カバレッジについて具体的な数字が得られるため、ユニット・テストが職人芸から科学にと変わります。Cobertura はテスト・カバレッジでの隙間を見つけ、それが直接、バグの発見につながります。コード・カバレッジを測定することによって、バグを発見し、修正するための情報が得られるため、より堅牢なソフトウェアを作ることができるのです。

---

# 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。