Are You Covered?

Keith D Gregory
Philly JUG
14 October 2009
Coverage
Tools
Lie
What is Coverage?

Measurement of how well your tests exercise your code

Metric: percent coverage

Coverage tools modify bytecode, inserting counters, then report result
public class Hello {
    public static void main(String[] argv) throws Exception {
        System.out.println("Hello, World");
    }
}

public static void main(java.lang.String[]) throws java.lang.Exception {
    getstatic #19; //Field java/lang/System.out
    invokevirtual #27; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
    return
}

public static void main(java.lang.String[]) throws java.lang.Exception {
    getstatic #41; //Field $VR4019:[[Z
    iconst_1
    aaload
    astore_1
    getstatic #19; //Field java/lang/System.out
    ldc #25; //String Hello, World
    invokevirtual #27; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
    aload_1
    iconst_0
    iconst_1
    astore
    return
}
Levels of Coverage

Class / Method / Line

Obsolete

Basic Block

Inserts counters after every branch
Shows partial coverage of if / loop / ternary
# Reporting

## Coverage Report - Mozilla Firefox

### Coverage Report - All Packages

<table>
<thead>
<tr>
<th>Package</th>
<th># Classes</th>
<th>Line Coverage</th>
<th>Branch Coverage</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Packages</td>
<td>44</td>
<td>94%</td>
<td>93%</td>
<td>3.786</td>
</tr>
<tr>
<td>net.sf.practicalxml</td>
<td>9</td>
<td>92%</td>
<td>91%</td>
<td>3.333</td>
</tr>
<tr>
<td>net.sf.practicalxml.builder</td>
<td>10</td>
<td>94%</td>
<td>90%</td>
<td>1</td>
</tr>
<tr>
<td>net.sf.practicalxml.converter</td>
<td>3</td>
<td>77%</td>
<td>0%</td>
<td>0</td>
</tr>
<tr>
<td>net.sf.practicalxml.converter.bean</td>
<td>11</td>
<td>95%</td>
<td>97%</td>
<td>0</td>
</tr>
<tr>
<td>net.sf.practicalxml.converter.internal</td>
<td>17</td>
<td>95%</td>
<td>89%</td>
<td>0</td>
</tr>
<tr>
<td>net.sf.practicalxml.converter.json</td>
<td>5</td>
<td>97%</td>
<td>89%</td>
<td>0</td>
</tr>
<tr>
<td>net.sf.practicalxml.internal</td>
<td>1</td>
<td>97%</td>
<td>94%</td>
<td>4.75</td>
</tr>
<tr>
<td>net.sf.practicalxml.junit</td>
<td>1</td>
<td>84%</td>
<td>100%</td>
<td>2/0</td>
</tr>
<tr>
<td>net.sf.practicalxml.util</td>
<td>5</td>
<td>61%</td>
<td>100%</td>
<td>4/4</td>
</tr>
<tr>
<td>net.sf.practicalxml.xpath</td>
<td>9</td>
<td>97%</td>
<td>96%</td>
<td>158/164</td>
</tr>
<tr>
<td>net.sf.practicalxml.xpath.function</td>
<td>4</td>
<td>90%</td>
<td>100%</td>
<td>12/17</td>
</tr>
</tbody>
</table>

Done

Fiddler: Disabled
Tools

Cobertura

http://cobertura.sourceforge.net/
Plugins for Ant, Maven 2
Licensed under GPL

Emma

http://emma.sourceforge.net/
Plugins for Ant, Maven 1, Eclipse
Can instrument on-the-fly via classloader
Licensed under IBM CPL
When to Run?

Daily or after Significant Change

Every build is too often
“Just before release” is not enough
Run first thing in morning, or after lunch

IDE integration encourages usage

Useful habit: follow with FindBugs™
Coverage Beyond the Unit Test

Interactive (desktop) applications

  Instrumentation adds minimal overhead

  Goal: identify action invocations (class-level coverage sufficient)

In-container testing of web-apps

  Integration tests

  Check on manual QA
The Problem

Coverage tools can only tell you whether your tests exercise your code.

They won’t tell you about missing features or missed specifications.

The report can be misleading.

You can get 100% coverage without fully testing your code.
Misleading Reports

It’s easy to put counters in bytecode
It’s harder to tie back to Java

Particularly if Hotspot modifies that code

Different tools may report different coverage for same code

Example: switch statement
Independent Paths

How many paths through this code?

```java
public int myFunc(int a, int b, int c) {
    if (a > 5)        
        c += a;     
    if (b > 5)         
        c += b;     
    return c;        
}
```
Independent Paths

Every non-trivial program has independent paths

Number of paths increases geometrically with number of “ifs”

Multi-threaded programs add time dimension — infinite number of paths
Independent Path Identification

Truth Table

- Can become unmanageable
- Useful as checklist for high-level code

Cyclomatic Complexity

- Generated as part of Cobertura report
- Bob Martin’s “C.R.A.P.” index
Refactor Mercilessly

Goal: one decision per method

High-level tests simply verify that lower-level methods were called
Exceptions

Most tests verify “happy path”
Need tests for likely exceptions
Understand boundary conditions
Not all exceptions can/should be tested
  Configuration exceptions from third-party libraries
  Exceptions that are rethrown or logged
“That test, I do not think it tests what you think it tests”

Tests should be tied to specifications

Are any specifications sufficiently detailed?
What about TDD?

Formal Test Driven Development:
- Write minimal failing unit test
- Write minimal mainline code to pass this test
- Refactor and repeat

This guarantees 100% coverage!

Beware when removing duplicated code
- Coverage tools useful to find dead code
- Sometimes, different code does different stuff
What about Getters & Setters?

Should be tested as part of normal use
   Don’t write tests just to validate they work

If not, why are they there?
   Is your code accessing the underlying members?
   Or is the data never used?
Code That Can’t Be Tested

Unreachable code

Example: default switch clause for enum

Exceptions that require drastic effort to induce

Example: configuration exception for JDK XML parser

Should such code even exist?
Coverage tools are valuable
They clearly highlight code that isn’t tested / used

100% coverage is unreasonable
Defensive coding may create unreachable code

100% coverage is not enough
Tests need to validate, not simply exercise
The Real Bottom Line

There’s no substitute for a dedicated, thoughtful, test–infected programmer
For More Information