CSC 307 Lecture Notes Week 10
Introduction to Code Coverage

I. Milestone 10 Summary

A. Due 11:59PM Thurs 11 June

B. Deliverables:

1. finished implementation of requirements subset
2. JML and unit tests for 8 to 12 methods
3. 100% code coverage for tested methods

II. **Final exam.**

A. Cummulative

B. Both paper and computer-based

C. See the final exam overview handout for details.

III. **What is code coverage?**

A. It is a measure of how program code is covered for a given program execution.

B. Coverage is typically measured at the level of textual lines of code.

C. When a program is run to completion, the coverage measure states the percentage of program lines that are covered, i.e., executed, during the run.

D. If all lines of code are executed, coverage is 100%; if only half the lines are executed, coverage is 50%.

IV. **How code goes "uncovered".**

A. Lines of code can go unexecuted for a number of reasons, including the following.

1. *Uninvoked functions* -- code in a function body is not covered if the function is never called during a program run
2. *Untaken conditional branches* -- depending on the values assigned to program variables, not all alternative branches of conditional statements may be covered in a particular program run.
3. *Unexecuted loop bodies* -- if a loop test never evaluates to true the loop body will not be executed.

B. In a testing context, uncovered code means there are insufficient test cases to fully exercise the code being tested.

V. **Coverage Tool Resources**

A. See the `307/doc/` page.

B. And note that code coverage is NOT required for Milestone 7, though it will be for a later milestone.

VI. **Where code coverage fits into testing.**

A. Code coverage is used to ensure that black box tests are adequately cover code.

B. There are many different coverage measures.

C. The bottom line is to ensure *some measure* of coverage.

D. During and after a test execution run, the coverage measures are applied to determine how much of the tested code is covered.

E. What follows is a discussion of the different coverage measures, from weakest to strongest.

VII. **Code coverage measures.**

A. Function (method) coverage.

B. Statement coverage

C. Branch coverage

D. Decision coverage

E. Loop coverage

F. Define-use (d-u) coverage

G. All path coverage

H. Exhaustive coverage

VIII. **Here is a common example that will be used to illustrate the different types of coverage.**

```
public static int f(int i, int j) {
    int k;
    if (i > j) {
        i++;
        j++;
    }
    k = g(i,j);
    if ((k > 0) && (i < 100)) {
        i++;
        j++;
    }
    else {
        i++;
    }
    return i+j+k;
}

static int g(int i, int j) {
    return i-j+1;
}
```

IX. **Function coverage.**

A. Each function is called at least once.

B. Very large-grain measure.

C. Not adequate for final tests.

D. Can be done with one test case for function `f`.

X. **Statement coverage.**

A. Every statement is executed at least once.

B. Can be done with two test cases for `f`.

XI. **Branch coverage.**

A. The true/false direction of each branch is taken at least once, including branches that have no code in them, as in an else-less if statement.

B. This requires four test cases for `f`.

XII. **All path coverage**

A. Each distinct control path is traversed.

B. Requires four cases for `f`.

XIII. **Decision coverage**

A. The boolean logic of each condition is fully exercised.

B. Requires at least four cases in `f`.

XIV. **D-u coverage**

A. Cover every path for every variable between a definition of that variable (i.e., assignment) and a use of that variable, without an intervening definition.

B. D-u for i requires three paths in `f`.

C. D-u for j requires two paths in `f`.

XV. **Coverage tools.**

A. There are a number of code coverage tools available for Java, and other languages, links to which are in the 307 `doc` directory.

B. One of the best of the tools is *Cobertura*.

C. There is an example of Cobertura in 307/examples/cobertura

1. The example code is that shown above as the common example for the different coverage measures.
2. It can be run in conjunction with JUnit tests, providing an HTML report of the code covered when the tests are executed.
3. There is an `ant` script to build and run the example, which is invoked simply by typing "`ant`" on the command line.
4. The examples files are the following:
   - *`CoverageExample.java`* -- the code to be tested and measured for coverage.
   - *`CoverageExampleTest.java`* -- the JUnit tester for the code.
   - *`build.xml`* -- the ant build script
   - *`build.properties`* -- the properties file for the build, which defines file paths for the build.
5. The results of running the example are in these subdirectories:
   - *`reports/cobertura-html`* -- the coverage report
   - *`reports/junit-html`* -- a junit testing report
6. As these reports suggest, Coberatura is well integrated with JUnit, and produces result reports for both test coverage and the JUnit test execution.

D. You can modify the ant scripts for use in other contexts by changing the `fileset` list in `build.xml`, and the directory paths in `build.properties`.

E. Alternatively, you can use Coberatura in an IDE, per the IDE's conventions.

XVI. **Details of branch (aka, decision) coverage in Cobertura and other coverage tools**

A. When a testing tool such as Cobertura reports that branch coverage is not attained, the code can be white-box analyzed to determine test cases to add to meet the branch coverage requirement.

B. Specifically, test values must be added to exercise fully the boolean logic of all in conditional statements.

C. The use of a truth table can help in this task.

D. As an example, here is the truth table for the four alternatives in the boolean logic of the conditional statement "`if ((k > 0) && (i < 100)) ...`":

k > 0 i < 100 (k > 0) && (i < 100) i j Remarks

0	0	0	1	2	i < j means k <= 0
0	1	0	100	101	i < j means k <= 0
1	0	0	100	100	i >= j mean k > 0
1	1	1	2	1	i >= j mean k > 0

XVII. **Some mutations to illustrate different forms of test failure.**

A. Change line 22 of `CoverageExample.java` from

```
return i+j+k;
```

to

```
return i+j-k;
```

1. This is an example of a bug in the code not properly implementing what the code is supposed to do.
2. The result is the coverage tests all still succeed, but three out of the four test cases in `CoverageExampleTest.java` fail

B. Change line 55 of `CodeCoverageExampleTest.java` from

```
assertEquals(-2, ce.f(-2,-1));
```

to

```
assertEquals(-2, ce.f(2,-1));
```

1. This is an example of a bug in the testing code not properly testing code that is correct.
2. In particular, the expected results are not correct with respect to the test plan.
3. The unit tests fail but the coverage tests still succeed

C. Further change line 55 of `CodeCoverageExampleTest.java` from

```
assertEquals(-2, ce.f(2,-1));
```

to

```
assertEquals(-2, ce.f(-2,1));
```

1. This is another example of a bug in the testing code not properly testing code that is correct.
2. Here the inputs are not correct with respect to the test plan.
3. The unit tests succeed but the coverage tests still succeed

XVIII. **Recent research on code coverage.**

A. Some very interesting research results on test coverage are presented in the paper

"Test coverage and post-verification defects: A multiple case study"
by Audris Mockus, Nachiappan Nagappan, Trung T. Dinh-Trong,
Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement,
October 2009

B. It is available at the ACM digital library at

http://portal.acm.org/citation.cfm?
id=1671248.1671276&coll=ACM&dl=ACM&CFID=80887391&CFTOKEN=40233171

access to which is free from campus computers (or anywhere to ACM digital library subscribers).

C. The paper authors are from Microsoft research and Avaya, two very large companies.

D. Key observations and conclusions from the paper are the following (***emphasis*** mine):

1. "Despite dramatic differences between the two industrial projects under study we found that ***code coverage was associated with fewer field failures*** ... . This strongly suggests that code coverage is a sensible

and practical measure of test effectiveness."

2. "[They found] an ***increase in coverage leads to a proportional decrease in fault potential.***"
3. "Disappointingly, there is ***no indication of diminishing returns*** (when an additional increase in coverage brings smaller decrease in fault potential)."
4. "What appears to be even more disappointing, is the finding that additional ***increases in coverage come with exponentially increasing effort***. Therefore, for many projects it may be impractical to achieve complete coverage."

E. Bottom line -- more coverage means fewer bugs, but it costs to get there.