Today

Introduction to software testing
- Blackbox vs. whitebox testing
- Unit testing (vs. integration vs. system testing)
- Test adequacy
  - Structural code coverage
    - Statement coverage
    - Decision coverage
    - Condition coverage
  - Mutation analysis

Software testing

What can testing do, and what can’t it do?
Software testing can show the presence of defects, but never show their absence! (Edsger W. Dijkstra)
- A good test is one that fails because of a defect.

How do we come up with good tests?

Unit testing, integration testing, system testing

Unit testing
- Does each unit work as specified?
Integration testing
- Do the units work when put together?
System testing
- Does the system work as a whole?

Our focus: unit testing

Two strategies: black box vs. white box

Black box testing
- The system is a black box (can’t see inside).
- No knowledge about the internals of a system.
- Create tests solely based on the specification (e.g., input/output behavior).

White box testing
- Knowledge about the internals of a system.
- Create tests based on these internals (e.g., exercise a particular part or path of the system).
Software testing

Software testing can show the presence of defects, but never show their absence! (Edsger W. Dijkstra)

- A good test is one that fails because of a defect.

When should we stop testing if no (new) test fails?

Test effectiveness

Ratio of detected defects is the best effectiveness metric!

Problem
- The set of defects is unknowable

Solution
- Use a proxy metric, for example code coverage

Structural code coverage: live example

Average of the absolute values of an array of doubles

```java
public double avgAbs(double ... numbers) {
    // We expect the array to be non-null and non-empty
    if (numbers == null || numbers.length == 0) {
        throw new IllegalArgumentException("Array numbers must not be null or empty!");
    }
    double sum = 0;
    for (int i=0; i<numbers.length; ++i) {
        double d = numbers[i];
        if (d < 0) {
            sum -= d;
        } else {
            sum += d;
        }
    }
    return sum/numbers.length;
}
```

Average of the absolute values of an array of doubles

Control Flow Graph (CFG)

Statement coverage

- Every statement in the program must be executed at least once
- Given the control-flow graph (CFG), this is equivalent to node coverage
Structural code coverage: statement coverage

Entry point

if (numbers == null || numbers.length == 0) {
    throw new IllegalArgumentException("Array numbers must not be null or empty!");
}

double sum = 0;
for (int i=0; i<numbers.length; ++i) {
    double d = numbers[i];
    if (d < 0) {
        sum -= d;
    } else {
        sum += d;
    }
}
return sum/numbers.length;

Condition coverage vs. decision coverage

Terminology

- **Condition**: a boolean expression that cannot be decomposed into simpler boolean expressions.
- **Decision**: a boolean expression that is composed of conditions, using 0 or more logical connectors (a decision with 0 logical connectors is a condition).
- **Example**: if (a & b) {...}
  - a and b are conditions.
  - The boolean expression a & b is a decision.

Decision coverage (a.k.a. branch coverage)

- **Every decision** in the program must take on **all possible outcomes** (true/false) at least once
- **Given the CFG**, this is equivalent to edge coverage
- **Example**: if (a>0 & & b>0)
  - a=1, b=1
  - a=0, b=0

Condition coverage

- **Every condition** in the program must take on **all possible outcomes** (true/false) at least once
- **Example**: (a>0 & & b>0)
  - a=1, b=0
  - a=0, b=1
Structural code coverage: live example

Average of the absolute values of an array of doubles

```java
public double avgAbs(double ... numbers) {
  // We expect the array to be non-null and non-empty
  if (numbers == null || numbers.length == 0) {
    throw new IllegalArgumentException("Array numbers must not be null or empty!");
  }
  double sum = 0;
  for (int i=0; i<numbers.length; ++i) {
    double d = numbers[i];
    if (d < 0) {
      sum -= d;
    } else {
      sum += d;
    }
  }
  return sum/numbers.length;
}
```

Structural code coverage: condition coverage

Decision coverage vs. condition coverage

4 possible tests for the decision $a \parallel b$:
1. $a = 0, b = 0$
2. $a = 0, b = 1$
3. $a = 1, b = 0$
4. $a = 1, b = 1$

Neither coverage criterion subsumes the other!

Structural code coverage: subsumption

Given two coverage criteria A and B, A subsumes B if satisfying A implies satisfying B

- **Subsumption relationships:**
  - Does decision coverage subsume statement coverage?
  - Does decision coverage subsume condition coverage?
  - Does condition coverage subsume decision coverage?

Structural code coverage: subsumption

Given two coverage criteria A and B, A subsumes B if satisfying A implies satisfying B

- **Subsumption relationships:**
  - Decision coverage subsumes statement coverage
  - Decision coverage does not subsume condition coverage
  - Condition coverage does not subsume decision coverage

Code coverage: advantages

- Code coverage is easy to compute.
- Code coverage has an intuitive interpretation.

But, does coverage ensure effective testing?
Code coverage: drawbacks

- Code coverage does not require test assertions.
- Not all statements etc. are equally important.
- Coverage is not the same as behavior.

Are there any alternatives?

Mutation analysis: overview

Each mutant contains one small syntactic change.
Mutation analysis: overview

Assumption: Mutant detection rate is a good proxy for fault detection rate.

What does it mean for a test to fail on a mutant program?

Mutation analysis: example

Find a test case that detects the following mutant (i.e., passes on the original program but fails on the mutant)

Original program:
public int min(int a, int b) {
    return a < b ? a : b;
}

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>Original</th>
<th>Mutant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Mutant:
public int min(int a, int b) {
    return a;
}

Mutation analysis: another example

Find a test case that detects the following mutant (i.e., passes on the original program but fails on the mutant)

Original program:
public int min(int a, int b) {
    return a < b ? a : b;
}

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>Original</th>
<th>Mutant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Mutant:
public int min(int a, int b) {
    return a <= b ? a : b;
}

There is no such test that can detect the mutant...

The mutant is undetectable because it is equivalent to the original program!

Summary

- Testing is an important way to measure code quality
- Black-box testing
- White-box testing
- Coverage metrics
  - Statement
  - Condition
  - Decision
- Mutation-based metric

For more, read: 
"Are mutants a valid substitute for real faults in software testing?" in FSE 2014