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.

How do we come up with good tests?
### 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).

### 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**

### Unit testing

- A **unit** is the smallest testable part of the software system.
- **Goal**: Verify that each software unit performs as specified.
- **Focus**:
  - Individual units (not the interactions between units).
  - Usually input/output relationships.

### 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;
}
```

Executable example available on the course web site.

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

```java
if (a==null || a.length==0) {
    throw new IllegalArgumentException("Array a must not be null or empty!");
}
sum = 0
i = 0
while (i<a.length) {
    num = a[i];
    if (num < 0)
        sum -= num;
    else
        sum += num;
    ++i;
}
return sum/a.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 (aka 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: (a>0 & b>0)
  - a=1, b=1
  - a=0, b=0

Structural code coverage: decision coverage

```java
if (a==null || a.length==0) {
    throw new IllegalArgumentException("Array a must not be null or empty!");
}
sum = 0
i = 0
while (i<a.length) {
    num = a[i];
    if (num < 0)
        sum -= num;
    else
        sum += num;
    ++i;
}
return sum/a.length;
```
**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: condition coverage**

- **Entry point**
  - `a==null || a.length==0` is `false`
  - `sum = 0` is `false`
  - `i = 0` is `false`
- **i<a.length** is `false`
- **num < 0** is `false`
- **num = a[i]** is `false`
- **++i** is `false`

**Decision coverage vs. condition coverage**

4 possible tests for the decision \(a \mid b\):

<table>
<thead>
<tr>
<th></th>
<th>(a)</th>
<th>(b)</th>
<th>(a \mid b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>(a)</th>
<th>(b)</th>
<th>(a \mid b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- Satisfies condition coverage but not decision coverage
- Does not satisfy condition coverage but decision coverage

Neither coverage criterion subsumes the other!
Structural code coverage: subsumption

Given two coverage criteria A and B, A subsumes B iff 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?

Are there any alternatives?

Code coverage: drawbacks

- Code coverage does not require test assertions.
- Not all statements etc. are equally important.
Mutation analysis: overview

Program

Test suite

Generate mutants

Mutants

Each mutant contains one small syntactic change
Mutation analysis: overview

Assumption: Mutant detection rate is a good proxy for fault detection rate.
(https://people.cs.umass.edu/~rjust/publ/mutants_real_faults_fse_2014.pdf)

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:
```java
public int min(int a, int b) {
    return a < b ? a : b;
}
```

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

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

<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>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

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:
```java
public int min(int a, int b) {
    return a < b ? a : b;
}
```

Mutant:
```java
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!

Mutation analysis: 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;
}
```

Executable example available on the course web site.