CSE P 590
Beyond Coverage: Modern Testing and Debugging
Spring 2019

Coverage-based testing

April 16, 2019
Today

- Recap: git bisect
- 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
      - MCDC
- In-class exercise 2
Recap: git bisect

PollEv.com/renejust859
Recap: git bisect

- How could the developers improve the build or testing infrastructure to notice test failures in the future?

- Which git command can you use to undo a defect-inducing commit? Briefly explain what problem may generally occur when undoing a commit and what best practices mitigate this problem.

- Can you undo the defect-inducing commit that you identified using a git command?
Today

- Recap: git bisect

- 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
      - MCDC

- In-class exercise 2
Software testing vs. software debugging

```java
1 double avg(double[] nums) {
2     int n = nums.length;
3     double sum = 0;
4     
5     int i = 0;
6     while (i<n) {
7         sum = sum + nums[i];
8         i = i + 1;
9     }
10     
11     double avg = sum * n;
12     return avg;
13 }
```
Software testing vs. software debugging

Testing: is there a bug?

```java
@Test
public void testAvg() {
    double[] nums = {1.0, 2.0, 3.0};
    double actual = Math.avg(nums);
    double expected = 2.0;
    assertEquals(expected, actual, EPS);
}
```

testAvg failed: 2.0 != 18.0
Software testing vs. software debugging

```java
1 double avg(double[] nums) {
2     int n = nums.length;
3     double sum = 0;
4     int i = 0;
5     while (i<n) {
6         sum = sum + nums[i];
7         i = i + 1;
8     }
9     double avg = sum * n;
10    return avg;
11 }
```

**Testing: is there a bug?**

```java
@Test
public void testAvg() {
    double[] nums = {1.0, 2.0, 3.0};
    double actual = Math.avg(nums);
    double expected = 2.0;
    assertEquals(expected, actual, EPS);
}
```

testAvg failed: 2.0 != 18.0

**Debugging: where is the bug?**

**how to fix the bug?**
Software testing

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

- **Unit**: A unit is the **smallest testable part** of the software system (e.g., a method in a Java class).
- **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.
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: 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;
}
```
Statement coverage

- Every statement in the program must be executed at least once.
Statement coverage

Entry point

- a==null || a.length==0
  - true: throw new IllegalArgumentException(“Array a must not be null or empty!”)
  - false:
    - sum = 0
    - i = 0

- i<a.length
  - true:
    - num = a[i]
    - num < 0
      - true: sum -= num
      - false: sum += num
    - ++i
    - return sum/a.length
  - false: return sum/a.length

Normal exit
Exceptional exit
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.
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**
Decision coverage (aka branch coverage)

Entry point

- \(a == \text{null} \lor a.\text{length} == 0\)

  - \(true\): throw new IllegalArgumentException("Array a must not be null or empty!")
    - Exceptional exit
  - \(false\)

  - \(sum = 0\)
  - \(i = 0\)

- \(i < a.\text{length}\)

  - \(false\): return \(sum / a.\text{length}\)
    - Normal exit
  - \(true\)

    - \(num = a[i]\)
    - \(num < 0\)
      - \(false\): \(sum += num\)
    - \(true\)
      - \(sum -= num\)
      - ++\(i\)
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:** \((a>0 \& b>0)\)
  - \(a=1, b=1\)
  - \(a=0, b=0\)
Condition coverage

- **Every condition** in the program must take on all possible outcomes (true/false) at least once

- **Example:** \((a>0 \ & \ b>0)\)
  - \(a=1, \ b=0\)
  - \(a=0, \ b=1\)
Condition coverage

Entry point

a==null || a.length==0

true

throw new IllegalArgumentException("Array a must not be null or empty!")

false

sum = 0
i = 0

i<a.length

false

return sum/a.length

true

num = a[i]

false

num < 0

true

sum -= num

++i

false

sum += num

Normal exit

Exceptional exit
Structural code coverage: subsumption

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

- Subsumption relationships:
  - Does statement coverage subsume decision coverage?
  - Does decision coverage subsume statement coverage?
  - Does decision coverage subsume condition coverage?
  - Does condition coverage subsume decision coverage?
Decision coverage vs. condition coverage

4 possible tests for the decision $a \mid b$:

1. $a = 0, b = 0$
2. $a = 0, b = 1$
3. $a = 1, b = 0$
4. $a = 1, b = 1$

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:
  - Statement coverage does not subsume decision coverage
  - Decision coverage subsumes statement coverage
  - Decision coverage does not subsume condition coverage
  - Condition coverage does not subsume decision coverage
MCDC

- Every decision in the program must take on all possible outcomes (true/false) at least once.
- Every condition in the program must take on all possible outcomes (true/false) at least once.
- Each condition in a decision has been shown to independently affect that decision’s outcome.

(A condition is shown to independently affect a decision’s outcome by: (1) varying just that condition while holding fixed all other possible conditions or (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome.)

Test requirement for safety critical systems (DO-178B/C)
MCDC: an example

if (a | b)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Which tests (combinations of a and b) are required to satisfy MCDC?

MCDC
- Decision coverage
- Condition coverage
- Each condition shown to independently affect outcome
MCDC: an example

if (a | b)

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

MCDC
- Decision coverage
- Condition coverage
- Each condition shown to independently affect outcome
MCDC: another example

if (a || b)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

MCDC

- Decision coverage
- Condition coverage
- Each condition shown to independently affect outcome

Why is this example different?
MCDC: another example

if (a || b)

| a  | b  | a || b |
|----|----|--------|
| 0  | 0  | 0      |
| 0  | 1  | 1      |
| 1  | -- | 1      |
| 1  | -- | 1      |

MCDC
- Decision coverage
- Condition coverage
- Each condition shown to independently affect outcome

Short-circuiting operators may not evaluate all conditions.
MCDC: yet another example

if (!a) { ... if (a || b) ... }

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

MCDC
- Decision coverage
- Condition coverage
- Each condition shown to independently affect outcome

What about this example?
MCDC: yet another example

```java
if (!a) { ... if (a || b) ... }
```

| a | b | a || b |
|---|---|-------|
| 0 | 0 | 0     |
| 0 | 1 | 1     |
| X | X | X     |
| X | X | X     |

**MCDC**
- Decision coverage
- Condition coverage
- Each condition shown to independently affect outcome

Not all combinations of conditions may be possible.
Code coverage: advantages

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