Introduction to Software Testing

Mayur Naik
CIS 573 – Fall 2019
Software Development Today

Manager

Developer

Tester
A Typical Scenario

I’m done!

OK, calm down. We’ll slip the schedule. Try again.

It doesn’t compile!
A Typical Scenario

Developer: I’m done!

Manager: Let’s have a meeting to agree on the spec.

Tester: It does the wrong thing in half the tests.

Developer: No, half of your tests are wrong!
A Typical Scenario

I’m done!

Manager

Try again but please hurry up!

Developer

Tester

It still fails some tests we agreed on.
A Typical Scenario

Developer: I’m done!

Manager:

Tester: I’m done, too!

Manager: Oops, the world has changed. Here’s the new spec.
Key Observations

- **Specifications** must be explicit
- Independent development and testing
- Resources are finite
- Specifications evolve over time
The Need for Specifications

- Testing checks whether program implementation agrees with program specification.

- Without a specification, there is nothing to test!

- Testing a form of consistency checking between implementation and specification:
  - Recurring theme for software quality checking approaches
  - What if both implementation and specification are wrong?
Developer != Tester

- Developer writes implementation, tester writes specification

- Unlikely that both will independently make the same mistake

- Specifications useful even if written by developer itself
  - Much simpler than implementation
  - => specification unlikely to have same mistake as implementation
Other Observations

• Resources are finite
  => Limit how many tests are written

• Specifications evolve over time
  => Tests must be updated over time

• An Idea: Automated Testing
  => No need for testers!?
Outline of This Lesson

• Landscape of Testing

• Specifications
  – Pre- and Post- Conditions

• Measuring Test Suite Quality
  – Coverage Metrics
  – Mutation Analysis
Classification of Testing Approaches

- Manual
- Automated

- Black-Box
- White-Box
Classification of Testing Approaches

- Manual
- Automated
- Black-Box
- White-Box

Needs Program Analysis
Automated vs. Manual Testing

• Automated Testing:
  – Find bugs more quickly
  – No need to write tests
  – If software changes, no need to maintain tests

• Manual Testing:
  – Efficient test suite
  – Potentially better coverage
Black-Box vs. White-Box Testing

• Black-Box Testing:
  – Can work with code that cannot be modified
  – Does not need to analyze or study code
  – Code can be in any format (managed, binary, obfuscated)

• White-Box Testing:
  – Efficient test suite
  – Potentially better coverage
An Example: Mobile App Security

HttpPost localHttpPost = new HttpPost(...);
(new DefaultHttpClient()).execute(localHttpPost);

http://[...]search.gongfu-android.com:8511/[...]
The Automated Testing Problem

• Automated testing is hard to do

• Probably impossible for entire systems

• Certainly impossible without specifications
Overview of Program Properties

• **Assertions**
  - Implicit, e.g. p != null before each deref of p
  - Explicit, e.g. assert (y == 7)

• **Types**
  - int x, java.net.Socket s, ...

• **Type-State Properties**
  - Program must not read data from java.net.Socket until socket is connected

• **Pre- and Post- Conditions**
  - double sqrt(double x) { ... }

  Pre-condition: x >= 0
  Post-condition: return value res satisfies res * res == x

All these are **Safety Properties** (i.e. partial correctness)

**Liveness Properties**: termination, starvation freedom, ...
Pre- and Post-Conditions

• A **pre-condition** is a predicate
  – Assumed to hold before a function executes

• A **post-condition** is a predicate
  – Expected to hold after a function executes, whenever the pre-condition also holds
class Stack<T> {
    T[] array;
    int size;

    Pre: s.size() > 0
    T pop() { return array[--size]; }
    Post: s'.size() == s.size() - 1

    int size() { return size; }
}
More on Pre- and Post-Conditions

• Most useful if they are executable
  – Written in the programming language itself
  – A special case of assertions

• Need not be precise
  – May become more complex than the code!
  – But useful even if they do not cover every situation
Using Pre- and Post-Conditions

Does test input satisfy pre-condition?

Yes

Run test with input

No

Go to next test

Does test result satisfy post-condition?

Yes

Test passes

No

Test fails

Doesn’t help write tests, but helps run them
Write the weakest possible pre-condition that prevents any in-built exceptions from being thrown in the following Java function.

```
int foo(int[] A, int[] B) {
    int r = 0;
    for (int i = 0; i < A.length; i++) {
        r += A[i] * B[i];
    }
    return r;
}
```
QUIZ: Pre-Conditions

Write the weakest possible pre-condition that prevents any in-built exceptions from being thrown in the following Java function.

```java
int foo(int[] A, int[] B) {
    int r = 0;
    for (int i = 0; i < A.length; i++) {
        r += A[i] * B[i];
    }
    return r;
}
```

Pre: $A \neq null \land B \neq null \land A.length \leq B.length$
QUIZ: Post-Conditions

Consider a sorting function in Java which takes a non-null integer array A and returns an integer array B. Check all items that specify the strongest possible post-condition.

- B is non-null
- B has the same length as A
- The elements of B do not contain any duplicates
- The elements of B are a permutation of the elements of A
- The elements of B are in sorted order
- The elements of A are in sorted order
- The elements of A do not contain any duplicates
QUIZ: Post-Conditions

Consider a sorting function in Java which takes a non-null integer array A and returns an integer array B. Check all items that specify the strongest possible post-condition.

✔ B is non-null
✔ B has the same length as A
☐ The elements of B do not contain any duplicates
✔ The elements of B are a permutation of the elements of A
✔ The elements of B are in sorted order
☐ The elements of A are in sorted order
☐ The elements of A do not contain any duplicates
Executable Post-Condition

- B is non-null
  
  ```
  B != null;
  ```

- B has the same length as A
  
  ```
  B.length == A.length;
  ```

- The elements of B are in sorted order
  
  ```
  for (int i = 0; i < B.length-1; i++)
    B[i] <= B[i+1];
  ```

- The elements of B are a permutation of the elements of A
  
  ```
  // count number of occurrences of
  // each number in each array and
  // then compare these counts
  ```
How Good Is Your Test Suite?

- How do we know that our test suite is good?
  - Too few tests: may miss bugs
  - Too many tests: costly to run, bloat and redundancy, harder to maintain
How Good Is Your Test Suite?

• How do we know that our test suite is good?
  – Too few tests: may miss bugs
  – Too many tests: costly to run, bloat and redundancy, harder to maintain

• Two approaches:
  – Code coverage metrics
  – Mutation analysis (or mutation testing)
Code Coverage

• Metric to quantify extent to which a program’s code is tested by a given test suite

• Given as percentage of some aspect of the program executed in the tests

• 100% coverage rare in practice: e.g., inaccessible code
  – Often required for safety-critical applications
Types of Code Coverage

• Function coverage: which functions were called?

• Statement coverage: which statements were executed?

• Branch coverage: which branches were taken?

• Many others: line coverage, condition coverage, basic block coverage, path coverage, ...
QUIZ: Code Coverage Metrics

Test Suite:
foo(1, 0)

Statement Coverage:  
%

Branch Coverage:  
%

Give arguments for another call to foo(x,y) to add to the test suite to increase both coverages to 100%.

x =  y = 

```c
int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}
```
QUIZ: Code Coverage Metrics

Test Suite:
foo(1, 0)

Statement Coverage: 80%
Branch Coverage: 50%

Give arguments for another call to foo(x,y) to add to the test suite to increase both coverages to 100%.

x = 1  y = 1

int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}
Mutation Analysis

• Founded on “competent programmer assumption”: 
  
  *The program is close to right to begin with*

• Key idea: Test variations (mutants) of the program
  – Replace $x > 0$ by $x < 0$
  – Replace $w$ by $w + 1, w - 1$

• If test suite is good, should report failed tests in the mutants

• Find set of test cases to distinguish original program from its mutants
### QUIZ: Mutation Analysis - Part 1

Check the boxes indicating a passed test.

<table>
<thead>
<tr>
<th>Mutant 1</th>
<th>Test 1 assert: foo(0,1) == 0</th>
<th>Test 2 assert: foo(0,0) == 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x \leq y \rightarrow x &gt; y$</td>
<td>☐</td>
<td>☐</td>
</tr>
<tr>
<td>Mutant 2</td>
<td>☐</td>
<td>☐</td>
</tr>
</tbody>
</table>

| x <= y → x != y |

```java
int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}
```

Is the test suite adequate with respect to both mutants?

- [ ] Yes
- [ ] No

- Yes
- No

"xyz"
# QUIZ: Mutation Analysis - Part 1

Check the boxes indicating a passed test.

<table>
<thead>
<tr>
<th>Mutant 1</th>
<th>Test 1 assert: foo(0,1)==0</th>
<th>Test 2 assert: foo(0,0)==0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x &lt;= y → x &gt; y</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mutant 2</th>
<th>Test 1 assert: foo(0,1)==0</th>
<th>Test 2 assert: foo(0,0)==0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x &lt;= y → x != y</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int foo(int x, int y) {</th>
</tr>
</thead>
<tbody>
<tr>
<td>int z = 0;</td>
</tr>
<tr>
<td>if (x &lt;= y) {</td>
</tr>
<tr>
<td>z = x;</td>
</tr>
<tr>
<td>} else {</td>
</tr>
<tr>
<td>z = y;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>return z;</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Is the test suite adequate with respect to both mutants?

- Yes
- No ✔

---

**Testing assert statements:**

**Test 1 assert:**

- Expected: foo(0,1)==0
  - Passed: ✔
  - Failed: No

**Test 2 assert:**

- Expected: foo(0,0)==0
  - Passed: ✔
  - Failed: ✔

---

**Function body:**

```
int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}
```
### QUIZ: Mutation Analysis - Part 2

Check the boxes indicating a passed test.

<table>
<thead>
<tr>
<th>Mutant 1</th>
<th>Test 1 assert: foo(0, 1) == 0</th>
<th>Test 2 assert: foo(0, 0) == 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x &lt;= y → x &gt; y</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mutant 2</th>
<th>Test 1 assert: foo(0, 1) == 0</th>
<th>Test 2 assert: foo(0, 0) == 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x &lt;= y → x != y</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Give a test case which Mutant 2 fails but the original code passes.

```c
int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}

assert: foo(   ,   ) ==
```
QUIZ: Mutation Analysis - Part 2

Check the boxes indicating a passed test.

<table>
<thead>
<tr>
<th>Mutant 1</th>
<th>Test 1 assert: foo(0,1)==0</th>
<th>Test 2 assert: foo(0,0)==0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x &lt;= y → x &gt; y</td>
<td></td>
<td>✔</td>
</tr>
</tbody>
</table>

| Mutant 2          |                             |                             |
| x <= y → x != y   | ✔                           | ✔                           |

int foo(int x, int y) {
    int z = 0;
    if (x <= y) {
        z = x;
    } else {
        z = y;
    }
    return z;
}

Give a test case which Mutant 2 fails but the original code passes.

assert: foo(1, 0) == 0
A Problem

• What if a mutant is equivalent to the original?

• Then no test will kill it

• In practice, this is a real problem
  – Not easily solved
  – Try to prove program equivalence automatically
  – Often requires manual intervention
What Have We Learned?

- Landscape of Testing
  - Automated vs. Manual
  - Black-Box vs. White-Box

- Specifications: Pre- and Post- Conditions

- Measuring Test Suite Quality
  - Coverage Metrics
  - Mutation Analysis
Many proposals for improving software quality

But the world tests
  → 50% of the cost of software development

Conclusion: Testing is important