Introduction to Software Testing

Introduction

1. Software Testing
    - The process of checking the correctness of a piece of software
    - Goals:
        - Describe the relationship of software testing to development
        - Classify different testing methods
        - Identify the specifications
        - Measure the quality of testing conducted on a given piece of software

Software Development Today

1. Software Development
    - Team consists of at least one developer, tester, and manager
    - Developers develop, testers test, managers manage specifications and timelines

Key Observations

1. Specifications must be explicit
2. Development and testing are completed independently
3. Resources are finite
    - Not every bug can be caught
4. Specifications evolve over time

The Need for Specifications

1. Testing checks whether program implementation agrees with program specification
2. Without a specification, there is nothing to test!
3. Testing is 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

1. Developer writes implementation, tester writes specification
2. Unlikely that both will independently make the same mistake
3. Specifications are useful even if written by developer itself
    - Much simpler than implementation
    - Specification unlikely to have same mistake as implementation

Other Observations

1. Resources are finite
    - Limit how many tests are written
    - Can't check every possible use case
2. Specifications evolve over time
    - Tests must be updated over time
3. An idea: Automated Testing
    - Automate writing and running of tests
    - No need for testers?

Outline of This Lesson

1. Landscape of Testing

    - Compare/contrast costs and benefits
2. Specifications
    - Pre- and Post-Conditions
    - How do we specify the behavior of functions?
3. Measuring Test Suite Quality
    - Is a test suite doing its job?
    - Coverage Metrics
    - Mutation Analysis

Classification of Testing Approaches

1. Manual vs. Automated
    - Describes the amount of human intervention in the testing process
2. Black-Box vs. White-Box
    - Describes the amount of access the testing apparatus has to the tested program source code
    - Black-box: Tester is unaware of the implementation details
    - White-box: Tester has knowledge of the program internals
3. Hybrid approaches
    - Manual/automated and Black/White box testing are spectrums, not discrete
4. Testing approaches
    - Tester tinkers with app's GUI
        - Manual, black-box
    - Tester tinkers with GUI, but has access to source code
        - Manual, white-box
    - Use an automated approach such as a fuzzer to issue tap commands to random coordinates of the smartphone
        - Automated, black box
    - Feedback-directed random testing: Issues random commands that change in response to the feedback issued by the GUI
        - Automated, white box
    - Symbolic execution: Need to inspect the source code in order to test effectively
        - Static analysis
    - Dynamic analysis: Monitor the code as it is being tested

Automated vs. Manual Testing

1. Automated Testing
    - Find bugs more quickly
    - No need to write tests
    - If software changes, no need to maintain tests
2. Manual Testing
    - Efficient test suite
        - Computer test suites can be blaoted
    - Potentially better coverage
        - Not guaranteed
3. Semi-automated
    - Human specifies the format or grammar valid inputs to a program, so that the automated testing that follows does not waste resources generating tests that do not exercise any interesting functionality of the program

Black Box vs. White Box Testing

1. 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)

2. White-box Testing
   - Efficient test suite
   - Potentially better coverage

An Example: Mobile App Security

1. DroidKungFu malware
   - Third-party app store that requested permissions, then sent sensitive information to other servers
   - Black-box testing would detect this malware by merely starting the app and monitoring the network activity of the phone
   - White-box testing would require inspecting the source of binary code of the app
     - Find the call to the external web server

The Automated Testing Problem

1. Difficulties with automated testing
   - Automated testing is hard to do
     - Can't always test the code under all possible conditions
   - Probably impossible for entire systems
   - Certainly impossible without specifications

Pre- and Post-Conditions

1. Pre-condition
   - Predicate that is assumed to hold before a function executes
2. Post-condition
   - Predicate that is expected to hold after a function executes, whenever the pre-condition also holds

Conditions Example

```
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; }
}
```

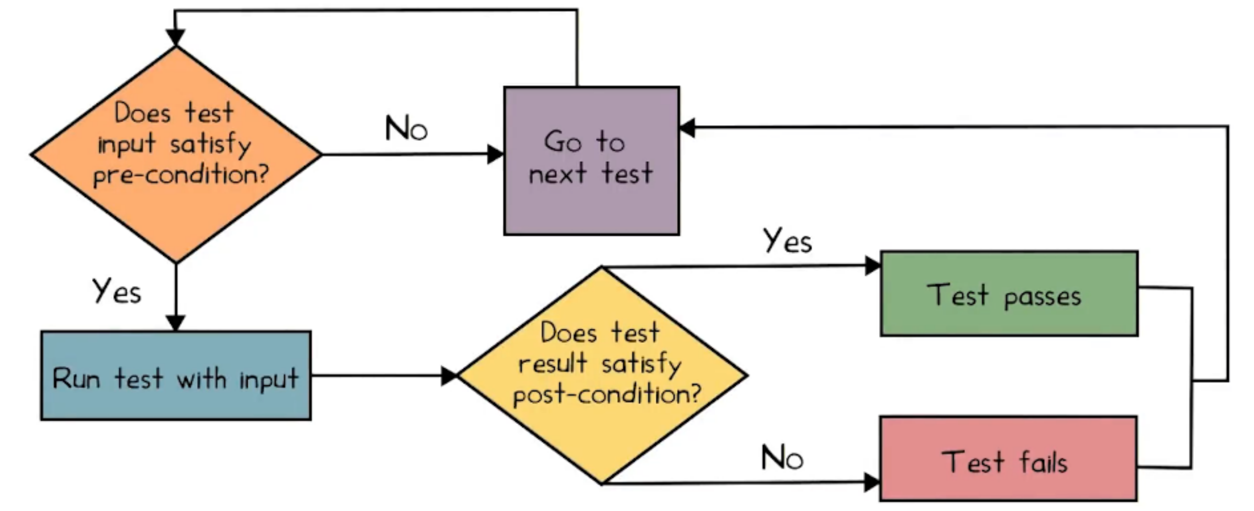
1. Frame condition
   - Implied conditions; nothing else changes about the state of the program beyond the pre- and post-conditions

More on Pre- and Post-Conditions

1. Most useful if they are executable
   - Written in the programming language itself
   - A special case of assertions
2. Need not be precise
   - May become more complex than the code!
   - But still useful even if they do not cover every situation

Using Pre- and Post-Conditions

1. Framework doesn't help write tests, but helps run them


Does test input satisfy pre-condition?
No
Go to next test
Yes
Run test with input
Does test result satisfy post-condition?
Yes
Test passes
No
Test fails

Pre- and Post-Condition Framework

Pre-Conditions

1. Write the weakest possible pre-condition that prevents any built-in exceptions from being thrown in the following Java function.
   - A != null && B != null && A.length <= B.length

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

Post-Conditions

1. 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-conditon.
   - B is non-null (true)
   - B has the same length as A (true)
   - The elements of B do not contain any duplicates (false)
   - The elements of B are a permutation of the elements of A (true)
   - The elements of B are in sorted order (true)
   - The elements of A are in sorted order (false)
   - The elements oF A do not contain any duplicates (false)

Executable Post-Condition

1. What would the post-condition look like in executable code?
   - B is non-null

    - B != null;
  - B has the same length as A
    - A.length == B.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 the number of occurrences of each number in each array and then compare these counts

How Good Is Your Test Suite?

1. 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
2. Two approaches:
   - Code coverage metrics
     - Has every line been tested?
   - Mutation analysis
     - Randomly mutate the program and run the same tests
     - If no tests fail, it indicates that the test suite may not be strong enough

Code Coverage

1. 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 rate in practive: e.g., inaccessible code
     - Often required in safety-critical applications

Types of Code Coverage

1. Function coverage: Which functions were called?
2. Statement coverage: Which statements were executed?
3. Branch coverage: Which branches were taken?
4. Many others: Line coverage, condition coverage, basic block coverage, path coverage

Code Coverage Metrics

1. Test suite:
   - foo(1, 0)
2. Calculate the statement and branch coverage:
   - Statement coverage: 80%
   - Branch coverage: 50%
3. Give arguments for another call to foo(x,y) to add to the test suite to increase both coverages to 100%.
   - X = 0
   - Y = 1

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

Mutation Analysis

1. Founded on the "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
    - It is possible to achieve high code coverage and still not uncover bugs

Mutation Analysis 1

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

	assert(foo(0,1) == 0)	assert(foo(0,0) == 0)
x <= y -> x > y	Fails	Passes
x <= y -> x != y	Passes	Passes

1. The test suite is not adequate with respect to both mutants

Mutation Analysis 2

1. Give a test case which mutant 2 fails but the original code passes.
    - assert(foo(1,0) == 0)

A Problem

1. What if a mutant is equivalent to the original?
    - No test will kill it
    - This makes it difficult to tell if we have a lack of robustness in our testing or if it's equivalent and we can ignore it
    - In practice, this is a real problem
        - Not easily solved
        - Try to prove program equivalence automatically
        - Often requires manual intervention

Conclusion

1. Landscape of Testing
    - Automated vs Manual
    - Black-box vs White-box
2. Specifications: Pre- and Post-Conditions
3. Measuring Test Suite Quality
    - Coverage metrics
    - Mutation analysis

Reality

1. Many proposals for improving software quality
2. But the world tests
   - 50% of the cost of software development
   - Some problems are inherently undecidable and can never be fully automated
3. Conclusion: Testing is important