Testing

Lawrence Angrave
About me
You're a CS Major at UIUC
... so testing should be obvious and easy?

"Returns true if and only if all strings are unique (only appear once in the array) and have no uppercase, false otherwise"

boolean isValid( String[] firstnames )
Poll "How many tests should we write?"

"returns true iff all unique & no uppercase, false otherwise"

boolean isValid(String[] firstname)

A) 3
B) 4
C) 5
D) 6
E) 7+

(not graded)
You're a CS Major at UIUC
... so testing should be obvious and easy?

— "Returns true if and only if all strings are unique (only appear once in the array) and have no uppercase, false otherwise"

boolean isValid( String[] firstnames )

<table>
<thead>
<tr>
<th>Test data</th>
<th>Expect</th>
<th>Purpose of test</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ &quot;a&quot;, &quot;b&quot;, &quot;c&quot; }</td>
<td>true</td>
<td>normative values</td>
</tr>
<tr>
<td>{ &quot;a&quot;, &quot;b&quot;, &quot;c&quot; }</td>
<td>false</td>
<td>has a dupe</td>
</tr>
<tr>
<td>{ &quot;a&quot;, &quot;b&quot;, &quot;B&quot; }</td>
<td>false</td>
<td>has uppercase</td>
</tr>
</tbody>
</table>

Are my tests sufficient?
Does the implementation meet the specification?
You're a CS Major at UIUC
... so testing should be obvious and easy?

"Returns true if and only if all strings are unique (only appear once in the array) and have no uppercase, false otherwise"

boolean isValid(String[] firstnames)

<table>
<thead>
<tr>
<th>Test data</th>
<th>Expect</th>
<th>Purpose of test</th>
</tr>
</thead>
<tbody>
<tr>
<td>{&quot;a&quot;, &quot;b&quot;, &quot;c&quot;}</td>
<td>true</td>
<td>normative values</td>
</tr>
<tr>
<td>{&quot;a&quot;, &quot;b&quot;, &quot;b&quot;}</td>
<td>false</td>
<td>has a dupe</td>
</tr>
<tr>
<td>{&quot;a&quot;, &quot;b&quot;, &quot;B&quot;}</td>
<td>false</td>
<td>has uppercase</td>
</tr>
</tbody>
</table>

Are my tests sufficient?
Does the implementation meet the specification?
Tests should be *mean* and *tricky*.
Time to get *dirty* ...

- My bad implementation has a *fault*, which must be discovered by your unit tests; at least one unit test must fail.
- $100 and *bragging rights* to the first student to show me a winning set of tests: Show me *in person* all of your Junit4 tests (or test case data). These must all pass in < 1 day for correct implementations of `isValid` (and at least one test must fail with an assertion failure for my implementation)

- No I won't show you my code… that would make it too easy.
- I'll read your unit tests/data to see if the *fault* is detected.
- Judge's decision (=my decision) is final. Must be currently enrolled in CS126 to play…
# So what are they?

<table>
<thead>
<tr>
<th>netids</th>
<th>Expect</th>
<th>Purpose of test</th>
</tr>
</thead>
<tbody>
<tr>
<td>{&quot;a&quot;,&quot;b&quot;,&quot;c&quot;}</td>
<td>true</td>
<td>normative example</td>
</tr>
<tr>
<td>{&quot;a&quot;,&quot;b&quot;,&quot;b&quot;}</td>
<td>false</td>
<td>has a dupe</td>
</tr>
<tr>
<td>{}</td>
<td>true</td>
<td>has uppercase</td>
</tr>
</tbody>
</table>

```
null
{...,null,...} {"aB"
{"A", "A"}
{"", "..."}
{"😊"}
```
Definitions:

- Programmer makes a **mistake**.
- **Fault** (defect, bug) appears in the program.
- The program **fails** (based on test oracles) during execution.
- An **error** is the difference between computed, observed, or measured value or condition and true, specified, or theoretically correct value or condition.
- **Testing** is detecting errors.
- Debugging is a means of diagnosing and correcting the root causes of errors that have already been detected.
Definitions: Mistake, Fault, Failure, Error

- Programmer makes a **mistake**.
- **Fault** (defect, bug) appears in the program.
- The program **fails** (based on **test oracles**) during execution.
- An **error** is the difference between computed, observed, or measured value or condition and true, specified, or theoretically correct value or condition.
- Testing is detecting errors.
- Debugging is a means of diagnosing and correcting the root causes of errors that have already been detected.
Why Test?

- Improve quality - find faults
- Measure quality
  - Prove there are no faults? (Is it possible?)
  - Determine if software is ready to be released
  - Determine what to work on
  - See if you made a mistake
- Learn the software
Definitions

- Which kind of testing is “specification testing”

A) Black box testing
B) White box testing
C) Grey box testing
Black box testing

- Tests the functionality of the system by observing its external behavior
- Tests against the specification
- No knowledge of how the code goes about meeting the goals
- Usually impossible to thoroughly exercise all inputs
Black box testing exercise II

- A function needs to be created such that given an integer value
  - it outputs 0 when the integer value is 8
  - it outputs 1 when the integer value > 8
  - It outputs -1 when the integer value < 8

- What would be your black box tests?
Tests

Max Value
Min Value

-1
0
-8
7
9

100,000,000
88
-42

f(int) null array

2^32 all
TESTING NUMERICAL CODE
Numerical testing – Write dirty tests based on common errors

- screenX = worldX / (worldZ + cameraPosition)
  - Divide by zero

- mergesort(data, lo, hi) {
  int midpoint = (lo + hi) / 2;
  - Overflow

- count --;
  - Underflow

- Floating point issues (precision, accuracy, …)
Numerical testing: Write dirty tests based on common errors

- NaNs and handling missing values
- Incorrect variable choice $m[i][j] = p[j][i] + q[i][k]$;
- Off-by one and boundary errors, $m[i-1][j-1] + m[i+1][j+1]$
- Physical units
Bag of Testing Tricks

- Equivalence Testing
- Error Guessing
- Boundary Analysis
- Classes of Good Data
- Classes of Bad Data
Black Box exercise III

- Construct tests for a piece of code that
  - Accepts an array of integers
  - returns false if anywhere in the array is there a number that is smaller than the number immediately before it in the array; otherwise returns true.

- Write your test cases and the expected return value

<table>
<thead>
<tr>
<th>Test data</th>
<th>Expect</th>
<th>Notes/Purposes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Blackbox vs Whitebox example

/* Sorts a list of values in reverse order. */
void reversesort(double[] values);

double[] data= {1,2,3};
double[] expected = {3,2,1};
reversesort( data );
assertArrayEquals(expected, data);
**Test cases?**

/* Sorts a list of values in reverse order. */

```c
void sortDescending(double[] values);
```

<table>
<thead>
<tr>
<th>Test Data</th>
<th>Notes/Purpose?</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1,2,3}</td>
<td></td>
</tr>
</tbody>
</table>
How do your tests change?

/* Sorts an array in reverse order */
void sortDescending(double(double[] values) {
    if( values == null) return; // No-op

    // For tiny lists insertion sort beats quicksort
    if( values.length < 30 ) {
        ... code for reversed insertion sort ...
        return;
    }

    // use existing quicksort then reverse result
    quicksort( values, 0, values.length - 1);
    reverse(values);
}
CODE COVERAGE
White box testing

- Test using knowledge about the inner workings of the code
- Test paths through the code
  - Conditionals, loops, etc.
- Can use coverage as a metric
  - Method vs. Statement vs. Decision/Branch vs. Condition
  - Coverage, however, is better for evaluating tests than for creating them
- Impossible to thoroughly exercise all possible complete paths
- Can be practical if a limited number of “important” paths are evaluated
100% Method Coverage

- All methods in all classes have at least been called
- Test case 1: Foo(0, 0, 0, 0, 0) = 0.0
- float foo (int a, b, c, d, e) {
  if (a == 0) {
    return 0.0;
  }
  int x = 0;
  if ((a==b) OR ((c==d) AND bug(a))) {
    x =1;
  }
  e = 1/x;
  return e;
}
100% Statement Coverage

- All lines in a method have been executed
- Test case 1 & 2: Foo(0,…) and Foo(1, 1, 1, 1, 1) = 1.0
- float foo (int a, b, c, d, e) {
  if (a == 0) {
    return 0.0;
  }
  int x = 0;
  if ((a==b) OR ((c==d) AND foo(a))) {
    x = 1;
  }
  e = 1/x;
  return e;
}
100% Branch/Decision Coverage

- Tests for if (evaluates to true) and if (evaluates to false)
- Test case 4: Foo(1, 2, 1, 2, 1) ← Found bug - division by zero!
- float foo (int a, b, c, d, e) {
  if (a == 0) {
    return 0.0;
  }
  int x = 0;
  if ((a==b) OR ((c==d) AND foo(a))) {
    x = 1;
  }
  e = 1/x;
  return e;
}
100% Condition Coverage

- All sub-expression predicates have been true and false
- Test case 4: Foo(1, 2, 1, 1, 1) ⇐ division by zero
- float foo (int a, b, c, d, e) {
  if (a == 0) {
    return 0.0;
  }
  int x = 0;
  if (((a==b) OR ((c==d) AND foo(a))) {
    x =1;
  }
  e = 1/x;
  return e;
}
Clicker Q

100% Condition coverage implies … (select best answer)
A) 100% Class coverage
B) 100% Method coverage
C) 100% Decision coverage
D) All of the above
Testing paths

- if (A && B) ...
  - has four cases, not just two; it has 3 paths

- Write a test cases for loops such that you:
  - Don’t go through the loop at all
  - Go through the loop once
  - Go through the loop many times
Black-box vs. White-box

- **White box** - look at code to write test
  - Tests are based on code
  - Better for finding crashes, out of bounds failures, file not closed failures
  - Better at finding faults of extra logic

- **Black box** - don’t look at code to write test
  - Tests are based on specifications
  - Better at telling if program meets spec
  - Better at finding faults of omission
Test First or Test Last?

A) Test First
B) Test Last
Test First

- Detect defects earlier (cheaper)
- Forces understanding of the requirements before you start coding
- Identifies problems with the requirements earlier
- No more effort to test first

- A tenet of eXtreme Programming (XP)
  - A design technique, not a testing technique
  - Doesn’t find bugs, but eliminates them
  - Doesn’t measure quality, but improves it
When to write tests

- During requirements analysis
- During architectural design
- During component design
- During coding
- After all coding
- Bug fixes \(-\text{regression test}\)
Definitions

- The repetition of previously executed test cases for the purpose of finding defects in software that previously passed the same set of tests.

A) Integration Testing
B) Regression Testing
C) Component Testing
D) Unit Testing
E) System Testing
Definitions

- The execution of the software in its final configuration, including integration with other software and hardware systems. It tests for security, performance, resource loss, timing problems, and other issues that can’t be tested at lower levels of integration.

A) Integration Testing
B) Regression Testing
C) Component Testing
D) Unit Testing
E) System Testing
What kind of tests?

- **Manual**
  - Good for exploratory
  - Good for testing GUI
  - Manual regression testing is BORING

- **Automatic**
  - Test is a program
  - Test is created by a tool that records user actions
  - The only way to make testing efficient as well as effective is to automate as much as possible
Fuzz or Monkey testing

- See if random input will crash the program

- “If you for some reason decide to monkey test your app on your phone, and not an emulator....be very sure to restrict it to the package of your app...I set it up with 5000 clicks, and couldn't stop it, and it opened up all kinds of apps, such as my text messaging app, music player, etc., and did all kinds of havoc on my phone. Even with package restricted, it messed with settings through the pulldown notification center, etc. What a nightmare. “

- Successfully used to find errors with Linux command line tools, Android UI

- Also used by security researchers to find vulnerabilities
Testing in Java
JUnit4

- Open source Java testing framework for automated testing
- Widely used in industry
- Features:
  - Assertions for testing expected results
  - Test features for sharing common test data
  - Test suites for easily organizing and running tests
  - Graphical and textual test runners
- Primarily for unit and integration testing, not system testing
JUnit Test Fixtures

- A test fixture is the state of the test
  - Objects and variables that are used by more than one test
  - Initializations (*prefix* values)
  - Reset values (*postfix* values)

- Tests can use the objects without sharing the state

- Objects used in test fixtures should be declared as instance variables

- They should be initialized in a @Before method
  - JUnit runs them before every @Test method

- Can be deallocated or reset in an @After method
  - JUnit runs them after every @Test method
Other testing comments

- Tests shouldn’t call other tests
public or private?

package com.startup.gameon;

public class ComputeSoundFx {
    void compute(byte[] data, float reverb);
}

package
Java testing tip 1

class com.startup.gameon.ComputeSoundFx
    "package com.startup.gameon;
    public class ComputeSoundFx {

src/com/startup/gameon/ComputeSoundFx.java

- Put tests in same package but different directory structure-

tests/com/startup/gameon/ComputeSoundFxTest.java

package com.startup.gameon;
public class ComputeSoundFxTest