Testing
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
Testing vs. Debugging

- Testing is detecting errors
- Debugging is a means of diagnosing and correcting the root causes of errors that have already been detected.
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
Definitions

- The execution of a complete class, routine, or small program that has been written by a single programmer or team of programmers, which is tested in isolation from the more complete system.

A) Integration Testing  
B) Regression Testing  
C) Component Testing  
D) Unit Testing  
E) System Testing
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.
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
- Impossible to thoroughly exercise all inputs
Black box testing exercise

- A program needs to be developed so that given an integer value
  - it outputs 0 when the integer value is 0
  - it outputs 1 when the integer value > 0
  - It outputs -1 when the integer value < 0

- What would be your black box tests?

  Ø → Ø
  Integer.MAXVALUE → 1
  5 → 1
  1 → 1
  -1 → -1
  Integer.MINVALUE → -1
Bag of Testing Tricks

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

- 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 proposed test cases and the expected return value:

- For \( [-3, -3, -3] \) \rightarrow true
- For \( [-7, -3] \) \rightarrow true
- For \( [-2, 2, 23] \) \rightarrow true
- For \( [-2, 3, 4, 5] \) \rightarrow true
- For \( [-4, -2, -1, -13] \) \rightarrow true
- For \( (i > -200; i < 1000; i++) \) \rightarrow true
- For \( [-2, 1, 3, 4, 5] \) \rightarrow false
- For \( [-1, -2, -3, 3] \) \rightarrow false
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 paths
- Can be practical if a limited number of “important” paths are evaluated
100% Method Coverage

- All methods in all classes have 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 2: 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

- All predicates have been true and false
- Test case 3: Foo(1, 2, 1, 2, 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;
}
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;
}
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 twice
  - 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
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
JUnit

- 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
public class Stack {
    public String toString() {
        // EFFECTS: Returns the String representation
        // of this Stack from the top to the bottom.
        StringBuffer buf = new StringBuffer ("{");
        for (int i = size-1; i >= 0; i--) {
            if (i < (size-1)) {
                buf.append (", ");
            }
            buf.append (elements[i].toString());
        }
        buf.append ("}"");
        return buf.toString();
    }

    public boolean repOk() {
        if (elements == null) return false;
        if (size != elements.length) return false;
        for (int i = 0; i < size; i++) {
            if (elements[i] == null) return false;
        }
        return true;
    }
}
Stack Test Class: Fixtures

Pre-test setup method (prefix):

```java
private Stack stack;
// setUp method using @Before syntax
// @Before methods are run before each test
@Before
public void runBeforeEachTest() {
    stack = new Stack();
}
```

Post-test teardown method (postfix):

```java
// tear-down method using @After
// @After methods are run after each test
@After
public void runAfterEachTest() {
    stack = null;
}
```
Stack Test Cases

A problem with this test is that it actually combines four separate tests in one method.

Without automation, large tests have the advantage of reducing costs of running many tests.

With automation, small tests allow us to more easily identify failures …
Stack Test Cases (2)

```java
@Test
public void testRepOkA() {
    boolean result = stack.repOk();
    assertEquals(true, result);
}

@Test
public void testRepOkB() {
    stack = stack.push(new Integer (1));
    boolean result = stack.repOk();
    assertEquals(true, result);
}

@Test
public void testRepOkC() {
    stack = stack.push(new Integer (1));
    stack = stack.pop();
    boolean result = stack.repOk();
    assertEquals(true, result);
}

@Test
public void testRepOkD() {
    stack = stack.push(new Integer (1));
    stack.top();
    boolean result = stack.repOk();
    assertEquals(true, result);
}
```
Other testing comments

- Tests shouldn’t call other tests
More kinds of Testing

- Fuzz or Monkey testing
  - See if random input will crash the program
To Dos for Thursday

- Fix any problems with your tests
- Make sure all of your tests pass
- Read Ch. 32 (Self-documenting code)
- Learn the structure of javadoc comments
- Make sure your code is well documented
- Push your updated code to github
  - Make sure that you have a single repo that contains your code and tests
- Generate the javadoc documentation for your project
  - You’ll show this off at your code review