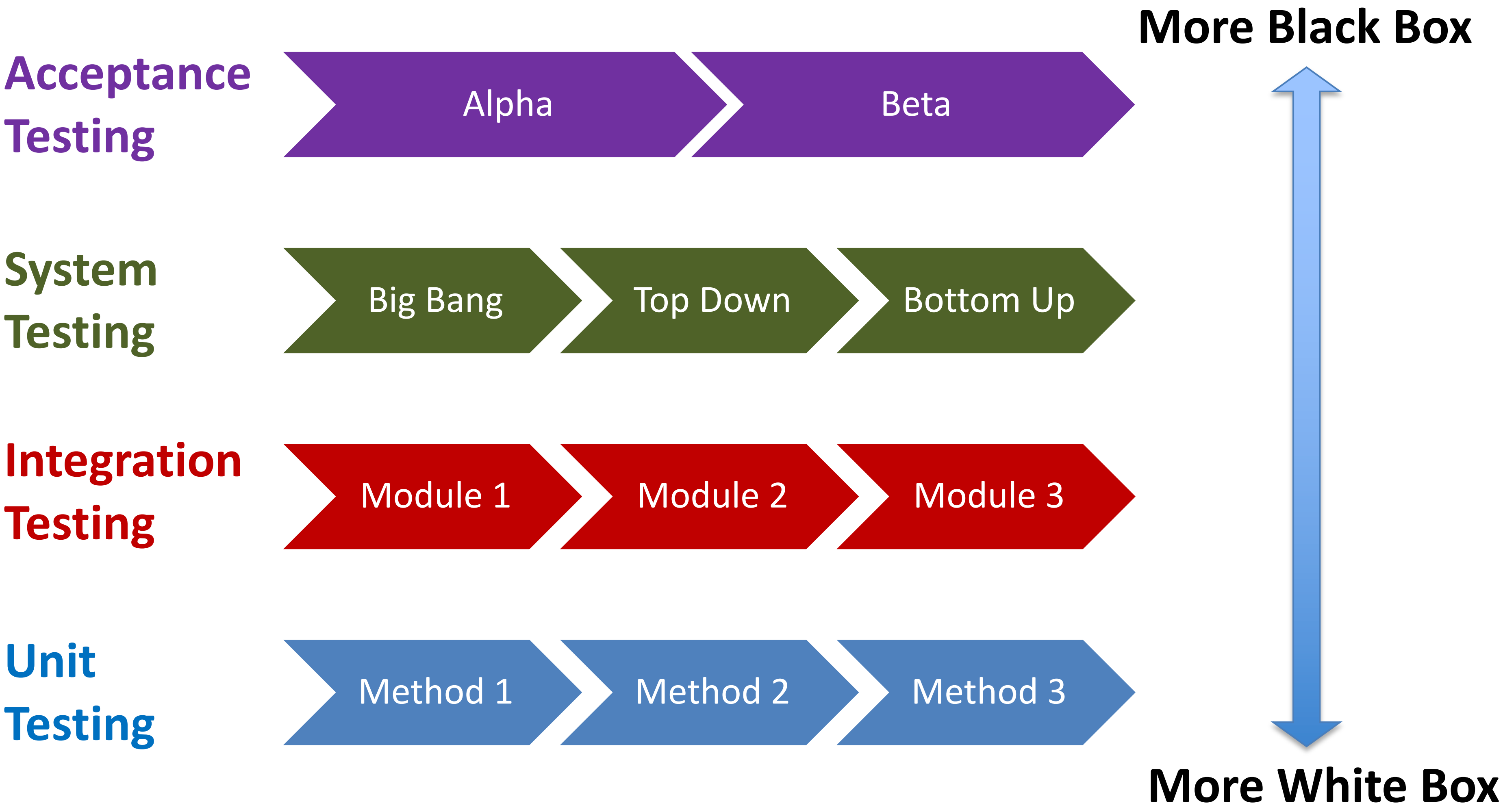
CS1632, Lecture 8: Unit Testing, part 1

Wonsun Ahn

What is unit testing?

- Unit testing: testing the smallest coherent "units" of code
  - Functions, methods, or classes
  - By directly invoking functions or methods
  - Necessarily white-box testing
- Goal: ensure the unit of code works correctly
  - Does NOT ensure the units taken together work correctly as a system
  - Very localized

The Four Levels of Software Testing



Acceptance Testing

System Testing

Integration Testing

Unit Testing

The Four Levels of Software Testing

- *Unit Testing*: Testing smallest unit of SW (typically a method)

- *Integration Testing*: Testing after integrating units into modules
  - A module in Java is analogous to a group of classes or package

- *System Testing*: End-to-end testing after integrating all modules
  - *Big Bang*: Testing at once after integrating all modules
  - *Top Down*: Testing incrementally by adding modules top-down
    - Uses *stubs* in place of not-yet-added leaf modules emulating those modules
  - *Bottom Up*: Testing incrementally by adding modules bottom-up
    - Uses a *driver* in place of not-yet-added root module calling the leaf modules
  - Why test incrementally? Easier to locate defect causing modules.

The Four Levels of Software Testing

- *(User) Acceptance Testing*: Checking SW is acceptable to user
  - Alpha Testing: Release to a select small group of users
    - Small group can be a select group of customers with high technical skill
    - Can be in-house, even the same development team (also called *dogfooding*)
    - Goal: To test and finalize the primary features of the SW
  - Beta Testing: Release to a broader set of users
    - *Closed Beta*: Also called *private beta*, only by invitation
    - *Open Beta*: Also called *public beta*, by anyone who wishes to participate
    - Goal: To ensure stability and security on various platforms and environments

Unit Testing Examples

- Testing that sort() method actually sorts elements
- Testing that formatNumber() method formats number properly
- Testing that passing in a null reference to a method which expects a valid object throws a NullPointerException
- Testing that passing in a string to a method which expects an integer throws a NumberFormatException

Who does Unit Testing?

- Usually done by the developer writing the code
- Another developer (esp. in pair programming)
- (Very occasionally), a white-box tester.

Why do Unit Testing?

1. Problems found earlier: no need to wait until system is built
2. Faster turnaround time: bug reporting overhead is not part of loop
   - Developer does the unit testing and can start debugging immediately
   - No need to wait for a tester to run test / file bug report / assign the bug
3. Developer understands issues with his/her code
   - Developer knows the code intimately and know where to find defects
4. "Living documentation"
   - Unit tests can be viewed as a documentation of expected behavior of the SW
   - Documentation is living because tests are verified regularly by running them against SW
5. Unit tests in sum total form a test suite
   - Test suite is run as regression test to find defects from changes with non-local impact
   - Unit test can discover defects due to changes in other units

What do Unit Tests Consist Of?

- A unit test is essentially a test case at the unit testing level
  - Same components: preconditions, execution steps, postconditions, …

- Anatomy of a unit test when implemented (e.g. using JUnit):
  - Preconditions: set up code (inits variables / data structures, …)
  - Execution Steps: one or more calls to unit tested method
  - Postconditions: assertions (checks postconditions are satisfied)
  - (Optional) tear down code (return to clean slate for next unit test)

A Unit Test Case for LinkedList.equals() method

- Preconditions:
  - Two linked lists with one node each
  - Nodes contain the integer value 1

- Execution Steps: Compare two lists with `equals()` method

- Postconditions: The `equals()` method SHOULD return true

A JUnit @Test Method is a Test Case

```
// Check that two LLs with one Node each with same val are equal
@Test
public void testEqualsOneNodeSameVals() {
    LinkedList<Integer> list1 = new LinkedList<Integer>();
    LinkedList<Integer> list2 = new LinkedList<Integer>();
    list1.addToFront(new Node<Integer>(new Integer(1)));
    list2.addToFront(new Node<Integer>(new Integer(1)));
    assertEquals(list1, list2);
}
```

- `assertEquals`: Invokes `equals()` method on arguments and asserts it returns true

A JUnit Class is a Test Plan

```
public class LinkedListTest {
   @Test public void testZeroList() { … }
   @Test public void testClearedList() { … }
   @Test public void testMultiList() { … }
   …
}
```

- Each `@Test` JUnit method is a test case
- Each JUnit class is a test plan
- Collection of JUnit classes is a test suite

Running A Test Suite

```
public class TestRunner {
   public static void main(String[] args) {
      ArrayList<Class> classesToTest = new ArrayList<Class>();
      // Add any JUnit test classes here
      classesToTest.add(LinkedListTest.class);
      // For all test classes, use JUnit to run them
      for (Class c: classesToTest) {
         Result r = JUnitCore.runClasses(c);
         // Print out any failures for this class.
         for (Failure f : r.getFailures()) {
            System.out.println(f.toString());
         }
      }
   }
}
```

More Linked List Test Cases

sample_code/ junit_example/LinkedListTest.java

Assertions = Postconditions Check

- When you think something "should" or "must" happen …
  - That is the EXPECTED BEHAVIOR or POSTCONDITION of the unit test
- When you execute the test by calling a method(s) …
  - That is when you'll find out the OBSERVED BEHAVIOR of your method
  - Either by retrieving return value(s) or side-effects of method

- Should assert EXPECTED BEHAVIOR == OBSERVED BEHAVIOR

JUnit assertions

- Some possible assertions using JUnit:
  - `assertEquals, assertArrayEquals, assertSame, assertNotSame, assertTrue, assertFalse, assertNull, assertNotNull, assertThat(*something*), fail(), ...`

- `assertSame(Object expected, Object actual)`: reference comparison
  - Compares two references with == operator rather than equals() method

- `assertThat(T actual, Matcher<T> matcher)`: a catch-all assertion
  - E.g. assertThat(“CS1632", anyOf(is(“cs1632"), containsString(“CS")));

- `fail()`: assertion that always fails
  - Why would you want an assertion that always results in test failure?
  - Maybe you shouldn’t have even gotten to that part of code

fail() example

```
// Check addToFront(null) results in IllegalArgumentException
@Test
public void testAddNullToNoItemLL() {
   LinkedList<Integer> ll = new LinkedList<Integer>();
   try {
      ll.addToFront(null);
      fail("Adding a null node should throw an exception");
   } catch (IllegalArgumentException e) {
   }
}
```

- Code execution never reaches fail() due to exception, as designed

Want more assertions?

- JUnit Javadoc reference:
  - http://junit.sourceforge.net/javadoc/org/junit/Assert.html

Test Fixture = Baseline Preconditions

- *Test fixture*: a fixed state used as a baseline precondition
  - Test cases in a test plan often need a common baseline precondition
  - Memory populated with a fixed set of objects
  - Database populated with a fixed set of entries
  - Hardware devices reinitialized to a fixed state

- In JUnit, implementable using @Before, @After
  - @Before annotation: Method executes before every @Test method
  - @After annotation: Method executes after every @Test method

Test Fixture Example

```
public class LinkedListTest {
    LinkedList<Integer> ll;
    Node<Integer>[] nodes;
    // Set up the test fixture before every @Test method
    @Before public void setUp() throws Exception {
        ll = new LinkedList<Integer>();
        nodes = new Node[10];
        for (int j = 0; j < 10; j++) {
            nodes[j] = new Node<Integer>(new Integer(j));
            ll.addToFront(nodes[j]);
        }
    }
    // Tear down the test fixture after every @Test method
    @After public void tearDown() throws Exception {}
    …
}
```

Test Fixture Example

```
public class LinkedListTest {
    LinkedList<Integer> ll;
    Node<Integer>[] nodes;
    @Before public void setUp() throws Exception { /* see previous slide */ }
    @After public void tearDown() throws Exception { /* see previous slide */ }
    @Test public void testClearList() {
        ll.clear();
        assertNull(ll.getFront());
    }
    @Test public void testDeleteFront() {
        ll.deleteFront();
        assertSame(ll.getFront(), nodes[8]);
    }
}
```

- Note: `ll` is reset with `node[9]`,`node[8]`,`node[7]`,…, `node[0]` before `testDeleteFront`

What values to test on method arguments?

- Ideally...
  - Each equivalence class
  - Both internal and boundary values

- And also both success and failure cases
  - *Success case*: inputs which follow the “happy path”
  - *Failure case*: inputs where method is expected to fail
  - Failure cases, as well as success cases, must follow requirements

Success Cases and Failure Cases

```
public String quack(int n) throws Exception {
  if (n > 0 && n < 10) {
    return "quack!".repeat(n);
  } else if (n >= 10) {
    throw new Exception("too many quacks");
  } else {   // n <= 0
    throw new Exception(“too little quacks");
  }
}
```

- Equivalence classes: {…, -2, -1, 0}, {1, 2, …, 9}, {10, 11, 12, …}
- Success cases: {1, 2, …, 9}
- Failure cases: {…, -2, -1, 0} + {10, 11, 12, …}

Public vs. Private Methods

- Two philosophies:
  - Test only public methods
  - Test every method – public and private
- Test only public methods
  - Private methods are tested as part of public methods anyway
  - Private methods get added/removed/changed more often
    - Why? Because they are not part of the public object interface
    - If we test them, we need to modify the test code every time!
  - Private methods may be difficult to test due to language/framework

Public vs. Private Methods

- Test every method – public and private
  - Public/private distinction is arbitrary – they are all units in your code
  - Unit testing means testing at the lowest level;
    Testing to the level of private methods adheres closer to the spirit
- Which philosophy to choose?
  - As everything in software QA, it depends ☺

Public Method Testing is Often Enough

```
class Bird {
   public int fly(int n) {
     return flapLeft(n) + flapRight(n);
   }
   // Tested as part of fly call.
   private int flapLeft(int n) { … }
   private int flapRight(int n) { … }
   // Never called! So no need to test anyway.
   private void urinate(double f) { … }
}
```

- A test of `fly` always tests `flapLeft` and `flapRight`
- Any private method not called in `fly` is in effect *dead code*

Where Public Method Testing is not Enough

```java
// Assume all the called methods are private
public boolean foo(boolean n) {
  if (bar(n) && baz(n) && beta(n)) {
    return true;
  } else if (baz(n) ^ (thud(n) || baa(n)) {
    return false;
  } else if (meow(n) || chew(n) || chirp(n)) {
    return true;
  } else {
    return false;
  }
}
```

- It’s a chore to even make sure each private method is tested
- If `foo` fails, hard to tell which private method has the defect

How can we test private methods?

- The programming language needs to allow it
- For Java, fortunately there is a way through something called *reflection*

```
class Duck {
  private int quack(int n) { … }
}
// Get method quack which has one argument of int type.
Method m = Duck.class.getDeclaredMethod("quack", int.class);
// Set method to accessible.
m.setAccessible(true);
// Pass arguments to invoke. 1st argument is always the instance.
Object ret = m.invoke(new Duck(), 5);
```

- Read Chapter 24 in Textbook for details

Now Please Read Textbook Chapter 13

- Also see sample_code/junit_example/LinkedListTest.java
  - For Mac/Linux: you can run all JUnit tests by “bash runTests.sh”
  - For Windows: you can run all JUnit tests by “runTests.bat”
  - Above script will invoke TestRunner to run test suite
- User manual:
  - https://junit.org/junit5/docs/current/user-guide/
- Reference Javadoc:
  - http://junit.sourceforge.net/javadoc/