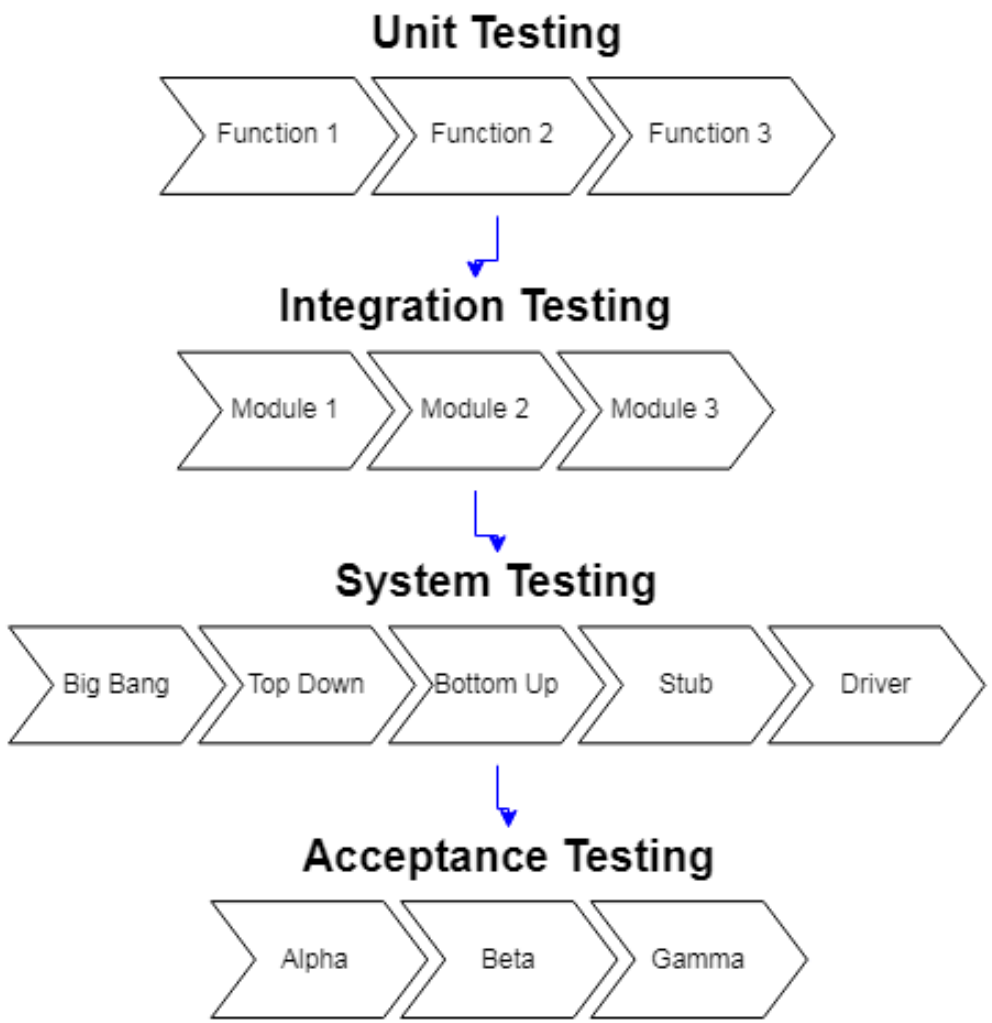
CS1632, Lecture 8: Unit Testing, part 1

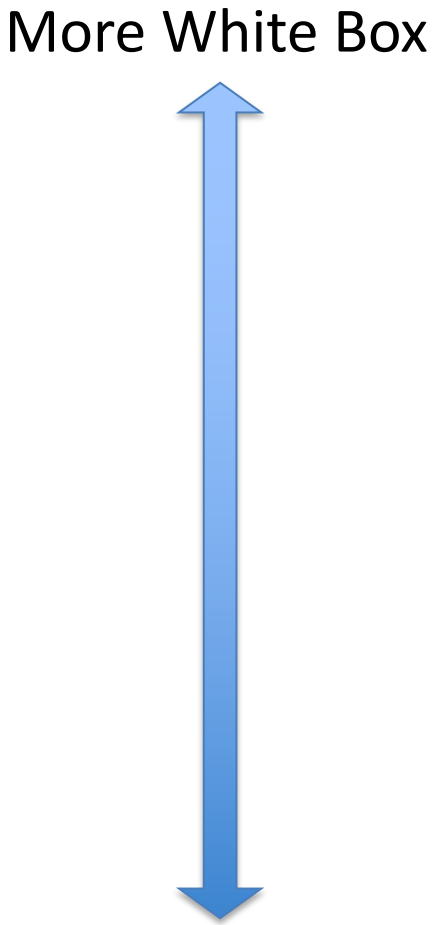
Wonsun Ahn

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





Examples

- Testing that sort() method actually sorts elements
- Testing that formatNumber() method formats number properly
- Testing that passing in a string to a function which expects an integer does not crash the program
- Testing that passing in a null reference throws an exception
- Testing that a send() and receive() methods exist on a class

Who does unit testing?

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

What’s the point?

1. Problems found earlier
2. Faster turnaround time
3. Developer understands issues with his/her code
4. "Living documentation“
5. Unit tests in sum total form a test suite
   - Running full test suite (e.g. as part of regression test) allows quick detection of defects due to code changes with non-local impact
   - Easy to zero-in on the failed unit if a unit test fails

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 result SHOULD be true

JUnit Implementation of Unit Test Case

```
// Check that two LLs with the same Node value with a single
node 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

More linked list test examples

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 that passing null to addToFront() results in an
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

What values to test on method arguments?

- Ideally...
  - Each equivalence class
  - Boundary values
  - Any other edge cases
- And also failure modes
  - Failure modes: inputs where method is expected to fail
  - Failing where it should is also part of its requirements

Equivalence Class / Boundary Value / Failure Mode

```
public int quack(int n) throws Exception {
  if (n > 0 && n < 10) {
    return 1;
  } 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, …}

Boundary values: 0, 1, 9, 10

Failure modes: {…, -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 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!
  - if used, private methods are tested as part of public methods anyway
  - 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 – you still want it all to be correct
  - 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 ☺

Where Public Method Testing may be Enough

```
class Bird {
   public int chirpify(int n) {
     return nirpify(n) + noogiefy(n + 1);
   }
   // Tested as part of chirpify call
   private int nirpify(int n) { … }
   private int noogiefy(int n) { … }
   // Never called! So no need to test!
   private void catify(double f) { … }
}
```

- If `chirpify` fails, it's either `nirpify` or `noogiefy` (if not self)

Where Public Method Testing is not Enough

```
// Assume all the called methods are complex
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, which 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 with int argument
Method m = Duck.class.getDeclaredMethod("quack", int.class);
// Set method to accessible
m.setAccessible(true);
// For instance methods, 1st argument is always instance
Object ret = m.invoke(new Duck(), 5);
```

- Read Chapter 24 in Textbook for details
- In this class, we will mostly limit ourselves to public methods

Now Please Read Textbook Chapter 13

- In addition, look carefully into: sample_code/junit_example/LinkedListTest.java
  - You can run all JUnit tests by executing runTests.sh
  - You will have to give execute permissions first (chmod +x runTests.sh)
  - Or invoke the shell explicitly (bash runTests.sh)
- User manual:
  - https://junit.org/junit5/docs/current/user-guide/
- Reference Javadoc:
  - http://junit.sourceforge.net/javadoc/