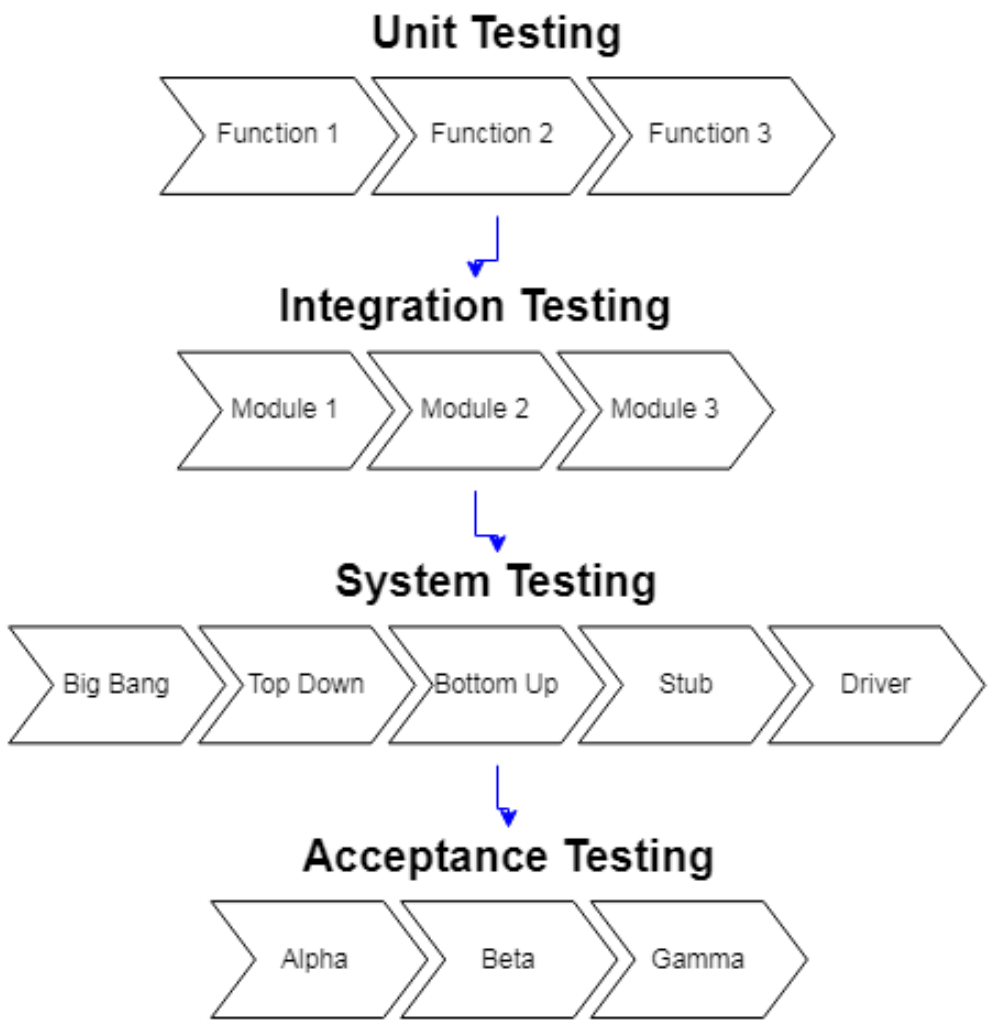
CS1632, Lecture 8: Unit Testing, part 1

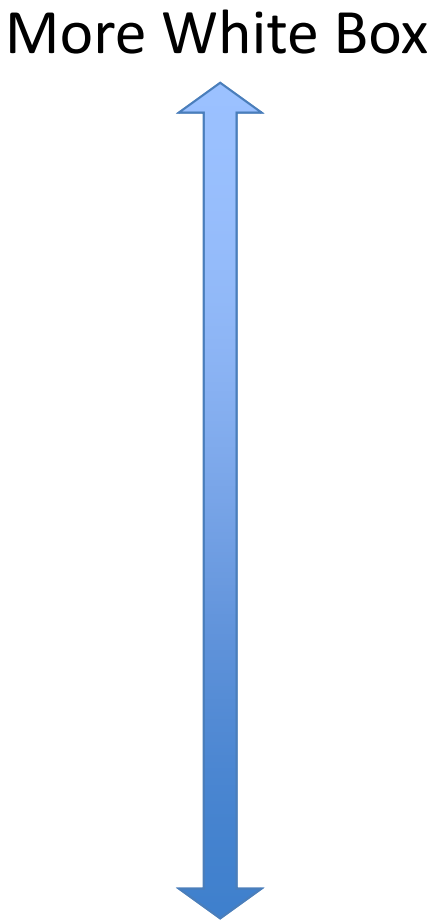
Wonsun Ahn

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)

Example of a Unit Test Case

- Preconditions: Two linked lists with one node each, where nodes contain the integer value 1
- Execution Steps: Compare the two lists with equality operator
- Postconditions: The result SHOULD be true

Unit test example

```
// Check that two LLs with the same Node value with a single
node are equal
@Test
public void testEqualsOneNodeSameVals() {
   LinkedList<Integer> ll11 = new LinkedList<Integer>();
   LinkedList<Integer> ll12 = new LinkedList<Integer>();
   ll11.addToFront(new Node<Integer>(new Integer(1)));
   ll12.addToFront(new Node<Integer>(new Integer(1)));
   assertEquals(ll11, ll12);
}
```

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

More linked list test examples

sample_code/ junit_example/LinkedListTest.java

Postconditions = assertions

- When you think "should" or "must", that is the assertion. It's what you're testing for.
- It's the EXPECTED BEHAVIOR of the unit test.
- When you execute the test, that's when you'll find out the OBSERVED BEHAVIOR.
- If the expected behavior matches the observed behavior, the test passes; otherwise it fails.

JUnit assertions

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

JUnit is not the only unit test framework out there!

- Not even for Java!
- But xUnit frameworks are common and easy to understand
  - C++: CPPunit
  - JavaScript: JSUnit
  - PHP: PHPUnit
  - Python: PyUnit
- Ideas should apply to other testing frameworks easily

What does a Good Unit Test Look Like?

- Reproducible on every run
- Independent of other tests
- Tests one thing at a time
- Tests only the unit

Good Unit Test: Reproducible on Every Run

- Test should either always pass or always fail. Why?
  - Otherwise, impossible to tell which build or version caused defect
  - The defect may have crept in long time ago but we were just lucky
- That means …
  - All preconditions must be set up correctly before running each test
  - There can be no random factor while running test
    - No randomness in the test itself (e.g. passing a random input value)
    - No randomness internal to the program (e.g. game with a die roll)
- How do we remove randomness internal to the program?!
  - Don't worry, we will learn in the next chapter ☺

Good Unit Test: Independent of Other Tests

- Tests should not be impacted by other tests. Why?

- We may choose to run a subset of tests in a test suite
  - If a test that this test depends on is not chosen …

- We may choose to run tests in a different order
  - We may even choose to run them in parallel!
  - Most unit testing frameworks allow parallel execution for faster completion

- Allows completion of test suite even in the event of failure
  - Even if a test fails, it does not impact any other tests

Good Unit Test:
Tests one thing at a time

- Do not test different test cases in a single test. Why?
  - If a test case fails (assertion fires), remaining test cases aren't tested
  - On test failure, hard to tell which test case failed

- Means you should call only one application method from test
  - The one that you are testing (unless it is unavoidable)

- If you use an "if..else" in a test, this is a code smell!
  - Value returned from a call used for something other than an assertion (for example, deciding which app method to call next, a no-no)

Good Unit Test:
Tests only the unit (and nothing else)

- Only test the code within the unit and nothing else. Why?
  - If test fails, you are sure the defect lies in the unit code
  - Otherwise, the defect may be in that "other" code you included
    - ☛ Defeats the entire purpose of unit testing!

- What if the unit (method) depends upon other code?
  - E.g. the method calls another method internally
  - Again, don't worry, we will learn in the next chapter ☺

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)