CSE 4513

Lec – 11

Debugging and Unit Testing

What is a Computer Bug

- ✓ a bug refers to an error, fault or flaw in any computer program or a hardware system.
- ✓ produces **unexpected results** or causes a system to behave unexpectedly.
- ✓ Any behavior or result that a program or system gets but it was not designed to do.
- ✓ From a developer perspective, bugs can be syntax or logic errors within the source code of a program.

WHAT IS DEBUGGING?

Things to think about:

- What caused the bug?
- How did you end up finding it?
- How could you have avoided the bug in the first-place?

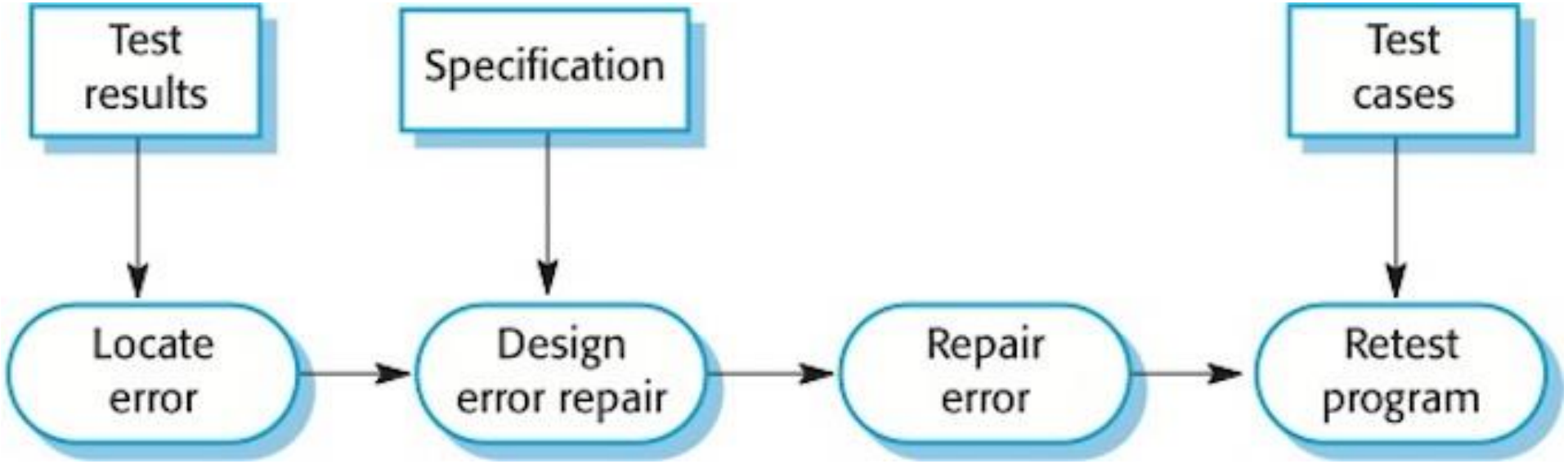
Debugging is a sanitization procedure consisting of:

- ✓ Preventing bugs in the first place, through good practices and assistive tooling.
- ✓ Detecting bugs when they first arise, through proper error handling and testing.
- ✓ Diagnosing and locating bugs, through the scientific method and interactive tooling.
- ✓ Treating bugs, through reasoned analysis and patient refactoring.

WHAT IS DEBUGGING?

Debugging is the process of finding and resolving defects or problems within a computer program that prevent correct operation of computer software or a system.

~Wikipedia



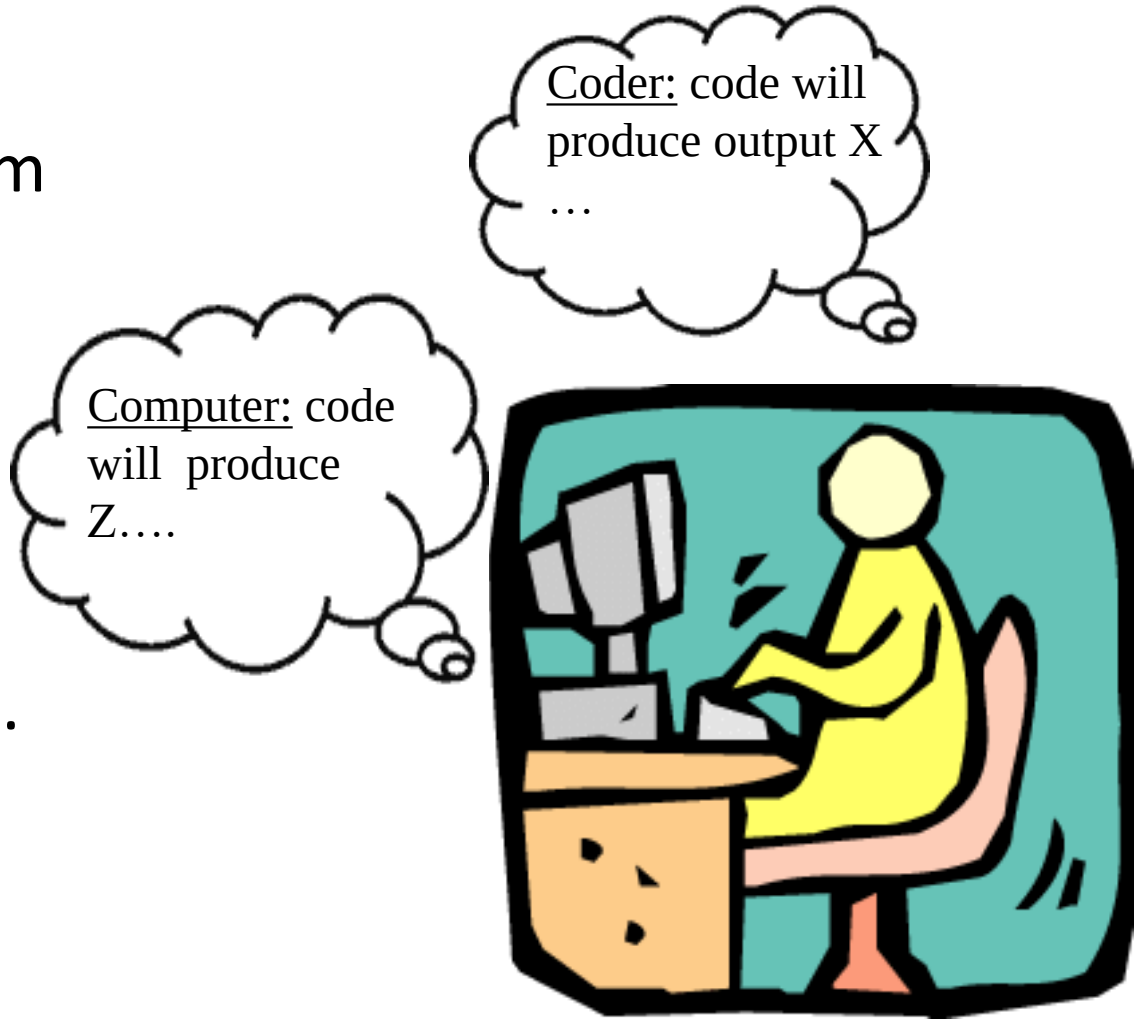
The debugging process

DEBUGGING PHILOSOPHY

Guiding Steps:

1) Think about why you believe the program should produce the output you expected.

2) Make assertions until you understand how your view differs from the computer's.



Strategy #1: Debug with Purpose

Don't just change code and “hope” you'll fix the problem!

Instead, make the **bug reproducible**, then use methodical “Hypothesis Testing”:

While(bug) {

1. Ask, what is the simplest input that produces the bug?
2. Identify assumptions that you made about program operation that could be false.
3. Ask yourself “How does the outcome of this test/change guide me toward finding the problem?”
4. Use pen & paper to stay organized!

}

Strategy #3: Focus on Recent Changes

If you find a NEW bug, ask:
what code did I change recently?

This favours:

- writing and testing code incrementally
- using ‘svn/git diff' to see recent changes
- regression testing (making sure new changes don't break old code).

STRATEGY #4: WHEN IN DOUBT, DUMP STATE

In complex programs, reasoning about where the bug is can be hard, and stepping through in a debugger time-consuming.

Sometimes its easier to just “dump state” and scan through for what seems “odd”.

Example:

Dumping all packets using tcpdump.

THE RISING COST OF FIXING A BUG

What do you think fixing a bug would cost on average?

At the XP Day 2009 conference in London, Google's **Mark Striebeck** reported on Google's estimates around the cost of delay in fixing defects.

- ✓ Google had estimated that it cost $5 to fix a bug immediately after a programmer had introduced it.
- ✓ Fixing the same defect would cost $50 if it escaped the programmer's eyes and was found only after running a full build of the project.
- ✓ The cost surged to $500 if it was found during an integration test, and to $5000 if it managed to find its way to a system test.

Considering these numbers, surely it's better to find out about such issues as soon as you possibly can!

WHAT IS UNIT TESTING

The testing that programmers do is generally called *unit testing.*
but we prefer not to use this term.

A *unit test* is a piece of code that invokes a unit of work and checks one specific end result of that unit of work. If the assumptions on the end result turn out to be wrong, the unit test has failed.

Unit testing is understood as **testing small parts of the code in isolation**.

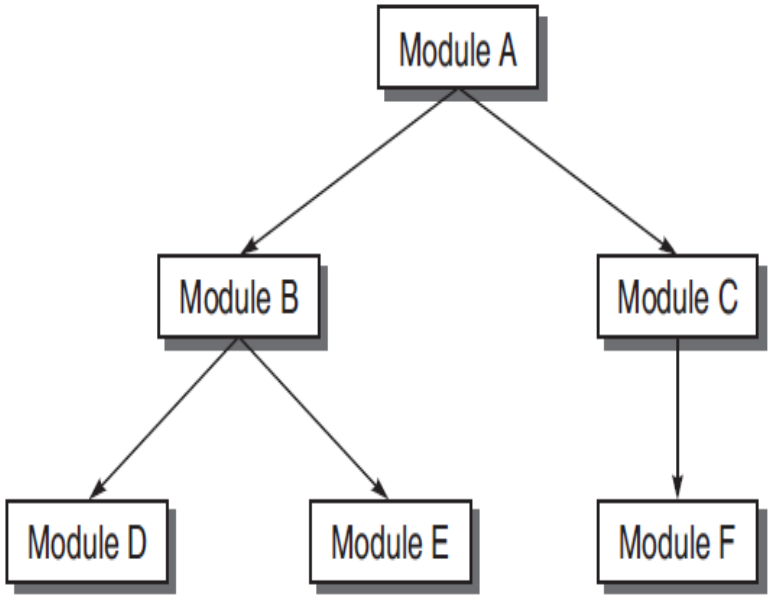
The unit under consideration might be getting some inputs from another unit or the unit is calling some other unit. It means that a unit is not independent and cannot be tested in isolation.

While testing a unit, all its interfaces must be simulated if the interfaced units are not ready at the time of testing the unit under consideration.

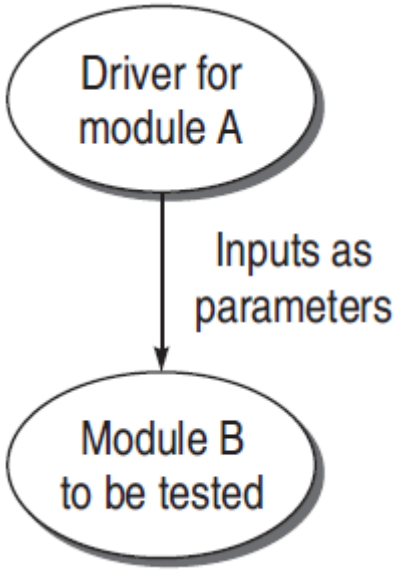
THING TO KNOW

Driver:

- Sometimes unit to be tested need input form other unit/module and that unit may not be implemented or under development
- In such a situation, we need to simulate the inputs required in the module to be tested.
- This module where the required inputs for the module under test are simulated for the purpose of module or unit testing is known as a driver module.



Suppose module A is not ready and B has to be unit tested. In this case, module B needs inputs from module A. Therefore, a **driver module** is needed which will simulate module A in the sense that it passes the required inputs to module B



THING TO KNOW

A test driver may take inputs in the following form and call the unit to be tested:

- ✓ It may hard-code the inputs as parameters of the calling unit.
- ✓ It may take the inputs from the user.
- ✓ It may read the inputs from a fi le.

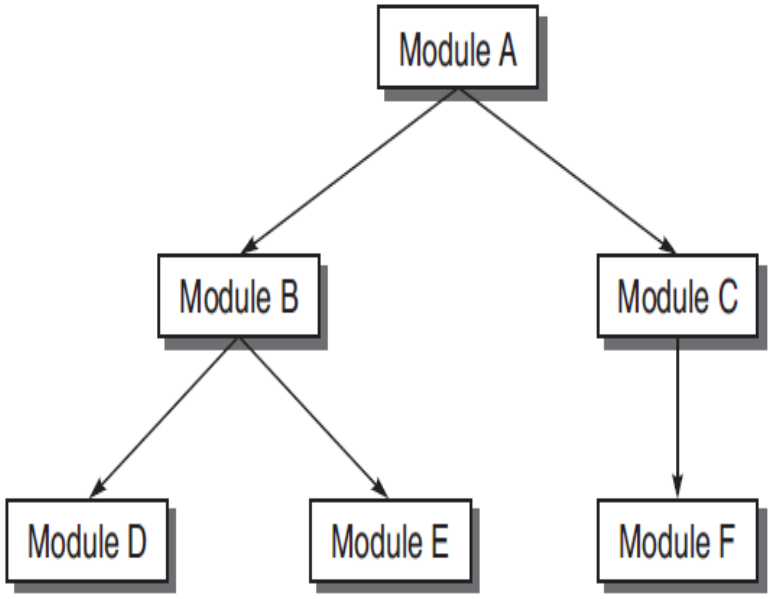
A test driver provides the following facilities to a unit to be tested:

- ✓ Initializes the environment desired for testing.
- ✓ Provides simulated inputs in the required format to the units to be tested.

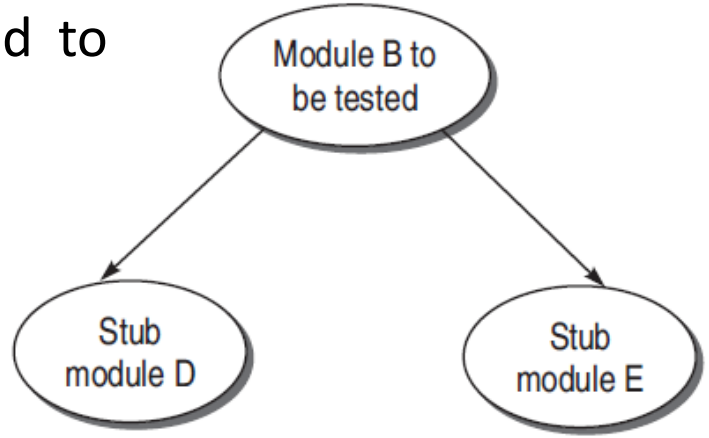
THING TO KNOW

Stubs:

- The module under testing may also call some other module which is not ready at the time of testing.
- these modules also need to be simulated for testing..
- For this dummy modules are prepared which is known as stubs.



Module B under test needs to call module D and module E. But they are not ready. So there must be some skeletal structure in their place so that they act as dummy modules in place of the actual modules. Therefore, **stubs** need to be designed for module D and module E



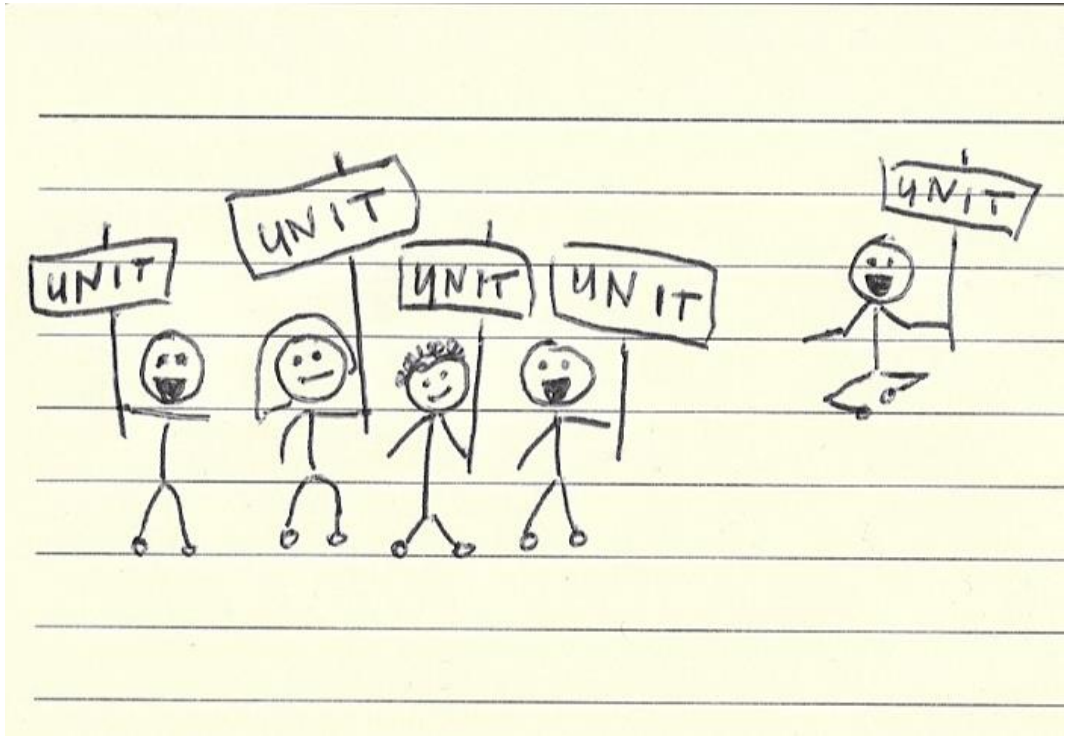
EXAMPLE

```
main()
{
    int a,b,c,sum,diff,mul;
    scanf("%d %d %d", &a, &b, &c);
    sum = calsum(a,b,c);
    diff = caldiff(a,b,c);
    mul = calmul(a,b,c);
    printf("%d %d %d", sum, diff, mul);
}
calsum(int x, int y, int z)
{
    int d;
    d = x + y + z;
    return(d);
}
```

(a) Suppose *main()* module is not ready for the testing of *calsum()* module.
Design a driver module for *main().*
(b) Modules *caldiff()* and *calmul()* are not ready when called in *main().*
Design stubs for these two modules.

How to identify a unit?

- ✓ It's entirely up to us!
- ✓ There are no hard and fast rules for this.
- ✓ We can consider a bunch of functions as one unit or test a single function as a separate unit.



But We can use a very simple rule:

If a unit proves difficult to test, then it's very likely that it needs to be broken down into smaller components.

PROPERTIES OF A GOOD UNIT TEST

A unit test should have the following properties:

- ✓ It should be easy to implement.
- ✓ Anyone should be able to run it at the push of a button.
- ✓ It should run quickly.
- ✓ It should be consistent in its results (it always returns the same result if you don't change anything between runs).
- ✓ It should have full control of the unit under test.
- ✓ It should be fully isolated (runs independently of other tests).
- ✓ When it fails, it should be easy to detect what was expected and determine how to pinpoint the problem.

Writing Good Tests

- Goal: to expose problems!
  - ✓ Assume role of an adversary
  - ✓ Failure == success
- Test boundary conditions
  - ✓ 0, Integer.MAX_VALUE, empty array
- Test different categories of input
  - ✓ positive, negative, and zero
- Test different categories of behavior
  - ✓ each menu option, each error message
- Test “unexpected” input
  - ✓ null pointer, last name includes a space
- Test representative “normal” input
  - ✓ random, reasonable values

The big Excuse

Tests are expensive to write

Yes, testing is expensive in most of industries: think about testing a home appliance, a drug or a new car…

- Test are costly
- They do **require extra effort**,
- but they improve quality, bring fast feedback, secure knowledge for newcomers…

YOUR FIRST UNIT TESTS

To check that two integers are equal, for instance, we could write a method that takes two integer parameters:

```
public void assertEquals(int a, int b) {
     assertTrue(a == b);
}
```

TESTING A SIMPLE METHOD

We'll start with a simple example, a single, static method designed to find the largest number in a list of numbers:

```
int Largest.largest(int[] list);
```

In other words, given an array of numbers such as [7, 8,9], this method should return 9.

That's a reasonable first test.

Think about more cases….

TESTING A SIMPLE METHOD

- How many tests did you come up with?

Your test cases should be as “What you pass in” → “What you expect”
[7, 8, 9] → 9
[8, 7, 9] → 9
[9, 7, 8] → 9

What happens if there are duplicates: [7, 9, 8, 9] → 9

What happens if there is only one number: [1] → 1

What happens with negative numbers: [-7, -9, -8, -9] → -7

To make all this more concrete, lets write a “largest” method and test it.

TESTING A SIMPLE METHOD

```
public class Largest  {
        /**
         * Return the largest element in a list.
         * @param list A list of integers
         * @return The largest number in the given list
         */
        public static int largest(int[] list) {
                int index, max=Integer.MAX_VALUE;
                for (index = 0; index < list.length-1; index++)
                        if (list[index] > max)  {
                                max = list[index];
                        }
                }
                return max;
        }
}
```

Now that we've got some ideas for tests, we'll look at writing these tests in Java, using the JUnit framework.

TESTING A SIMPLE METHOD

First, let's just test the simple case of passing in a small array with a couple of unique numbers. Here's the complete source code for the test class.

```
import junit.framework.*;
public class TestLargest extends TestCase {
        public TestLargest(String name) {
                super(name);
        }
        public void testSimple() {
                assertEquals(9, Largest.largest(new int[] {7,8,9}));
        }
}
```

TESTING A SIMPLE METHOD

Having just run that code, you probably saw an error similar to the following:

```
There was 1 failure:
1) testSimple(TestLargest)junit.framework.AssertionFailedError:
expected:<9> but was:<2147483647>
at TestLargest.testSimple(TestLargest.java:11)
```

Whoops! That is not expected. Why does it return such a huge number instead of 9?

oh, it's a small typo: `max=Integer.MAX VALUE` on line 11 should have been `max=0`. We want to initialize max so that any other number instantly becomes the next max. Let's fix the code, recompile, and run the test again to make sure that it works.

TESTING A SIMPLE METHOD

Next we'll look at what happens when the largest number appears in different places in the list first or last, and somewhere in the middle.

```
import junit.framework.*;
public class TestLargest extends TestCase {
  public TestLargest(String name) {
    super(name);
  }
  public void testSimple() {
    assertEquals(9, Largest.largest(new int[] {7,8,9}));
  }
  public void testOrder() {
    assertEquals(9, Largest.largest(new int[] {9,8,7}));
  }
}
```

it works☺

TESTING A SIMPLE METHOD

Now try it, with the mistake in the code (just add an assertion to the existing testOrder() method):

```
publ    public void testOrder() {
          assertEquals(9, Largest.largest(new int[] {9,8,7}));
   as     assertEquals(9, Largest.largest(new int[] {7,9,8}));
   as     assertEquals(9, Largest.largest(new int[] {7,8,9}));
}       }
```

```
There was 1 failure:
1) testOrder(TestLargest)junit.framework.AssertionFailedError:
   expected:<9> but was:<8>
   at TestLargest.testOrder(TestLargest.java:10)
```

Why did the test get an 8 as the largest number?

TESTING A SIMPLE METHOD

Why did the test get an 8 as the largest number?

It seems the code ignored the last entry in the list. Sure enough, another simple typo: the `for` loop is terminating too early.

Our code has:

```
for (index = 0; index < list.length-1; index++) {
```

But it should be one of :

```
for (index = 0; index <= list.length-1; index++) {
for (index = 0; index < list.length; index++) {
```

TESTING A SIMPLE METHOD

Now you can check for duplicate largest values:

```
public void testDups() {
  assertEquals(9, Largest.largest(new int[] {9,7,9,8}));
}
```

it works☺

Now the test for just a single integer:

```
public void testOne() {
   assertEquals(1, Largest.largest(new int[] {1}));
}
```

it works☺

We are almost done except the negative test

TESTING A SIMPLE METHOD

Now you can check for Negative values:

```
public void testNegative() {
  int [] negList = new int[] {-9, -8, -7};
  assertEquals(-7, Largest.largest(negList));
}
```

```
There was 1 failure:
1) testNegative(TestLargest)junit.framework.AssertionFailedError:
   expected:<-7> but was:<0>
   at TestLargest.testNegative(TestLargest.java:16)
```

How is the actual result zero?

Looks like choosing 0 to initialize max was a bad idea; what we really wanted was MIN VALUE, so as to be less than all negative numbers as well:

```
max = Integer.MIN_VALUE
```

TESTING A SIMPLE METHOD

Unfortunately, the initial specification for the method “largest” is incomplete, as it does not says what should happen if the array is empty.

We need to handle that…

Let's say that it's an error, and add some code at the top of the method that will throw a runtime-exception if the list length is zero:

```
public static int largest(int[] list) {
  int index, max=Integer.MAX_VALUE;
  if (list.length == 0) {
    throw new RuntimeException("Empty list");
  }
  ...
```

Now for the last test, we need to check that an exception is thrown when passing in an empty array.

TESTING A SIMPLE METHOD

```
public void testEmpty() {
   try {
      Largest.largest(new int[] {});
      fail("Should have thrown an exception");
   } catch (RuntimeException e) {
      assertTrue(true);
   }
}
```

it works☺

We started with a very simple method and come up with a couple of interesting tests that actually found some bugs.

Unit Testing Best Practices

- ✓ Unit Test cases should be independent. In case of any enhancements or change in requirements, unit test cases should not be affected.
- ✓ Test only one code at a time.
- ✓ Follow clear and consistent naming conventions for your unit tests
- ✓ In case of a change in code in any module, ensure there is a corresponding unit Test Case for the module, and the module passes the tests before changing the implementation
- ✓ Bugs identified during unit testing must be fixed before proceeding to the next phase in SDLC