CSC322

Lecture Note 6: Test First Programming

Dr. Victor Odumuyiwa

Learning Outcomes

At the end of this class, you should be able to:

- Write unit tests

Credit

Majority of the contents are from MIT 6.005 lecture note on test-first programming.

TEST-FIRST PROGRAMMING

Test-first programmimg

- Testing is only one part of a more general process called validation.

Assertion

- An assertion verifies an assumption of truth for compared conditions.
- The Assert class throws an AssertFailedException to signal a failure. This exception should not be captured. This exception is handled by the unit test engine to indicate an assert failure.

Regression Testing

- Running all your tests after every change is called regression testing
- Whenever you find and fix bugs
  - Store the input that elicited the bug
  - Store the correct output
  - Add it to your test suite

Why regression tests help

- Help to populate test suites with new test cases
  - Remember that a test is good if it elicits a bug, and every regression test did in one version of your code
- Protect against reversions that reintroduce bugs
- The bug may be an easy error to make (since it happened once already)

Test-first debugging

- When a bug arises, immediately write a test case for it that elicits it
- Once you find and fix the bug, the test case will pass and you'll be done

Unit Testing in C# (msdn.com)

A test method must meet the following requirements:

- The method must be decorated with the [TestMethod] attribute.
- The method must return void.
- The method cannot have parameters.

Creating test cases

- **Partition the Input Space**
  - input space is very large, but program is small
  - behavior must be the "same" for whole sets of inputs
- **Boundary testing**
  - include classes at **boundaries** of the input space
  - why? because bugs often occur at boundaries

Blackbox vs. glassbox(whitebox)

- Blackbox testing
  - choosing test data only from spec, without looking at implementation
- Glassbox testing
  - choosing test data with knowledge of implementation
  - must take care that tests don't *depend* on implementation details

Good tests should be modular -- depending only on the spec, not on the implementation

Coverage

- All-statements: is every statement run by some test case?
- All-branches: if every direction of an if or while statement (true or false) taken by some test case?
- All-paths: is every possible combination of branches – every path through the program – taken by some test case?

Method to be tested

```
public int Find(int x, int[] a)
        {
            for (int i = 0; i < a.Length; i++)
            {
                if (x == a[i])
                {
                    return i;
                }
            } return -1;
        }
```

Test case 1

```
public void findTestOnCase1()
        {
            var testObject = new BinarySearch();
            int x = 2;
            int[] a = { -1, 1, 3 };
            int expected = -1;
            int actual = testObject.Find(x, a);
            Assert.AreEqual<int>(expected,
actual);
        }
```