Unit Testing Review

DevBuild

Learning Objectives

- Define unit testing
- Explore unit testing theory
- Explore unit testing best practices
- Review common unit testing syntax
- Example on how to write testable methods

What is a unit test?

Unit testing is a form of testing that breaks down code into the smallest piece that can logically be isolated, or "a unit", to ensure the program is functioning as intended.

Black box testing: Testing that does not care about implementation, only input and intended output.

White box testing: Looks at implementation in attempt to achieve complete code coverage, requires access to source code

Why unit testing?

By writing tests first for the smallest unit, they can build up to comprehensive tests for a complex application. Using this bottom-up approach to testing makes integration testing much easier down the road.

Unit Testing Best Practices

A helpful acronym to remember for unit tests is F.I.R.S.T.:

- **Fast**: Some projects may have thousands of unit tests, so it important that each test is efficient and executes within milliseconds
- **Isolated:** Unit tests can be run in isolation having no dependencies on outside factors (i.e. file systems, databases, user input)
- **Repeatable:** Unit tests should be deterministic, always returning the same results between runs where no code is modified.
- **Self-Checking:** The test should detect if it passed or failed without human interaction
- **Timely:** The test should not take disproportionate time to write compared to the code being tested. If it is, consider a design that is more easily tested.

Naming conventions: Tests should be named in a consistent, relevant convention. You want anybody looking at your tests to immediately be able to understand what that specific test is testing.

Common Assertions:

- Assert.Equal(expected, actual): Verifies if two objects are equal using default comparator. Several overloaded versions of this exist.
- Assert.True(condition): Verifies the condition is true
- Assert.False(condition): Verifies the condition is false
- Assert.Throws<T>: Verifies the exact exception of type T is thrown
- Full suite of XUnit assertions: https://github.com/xunit/assert.xunit

Your unit tests should allow you to make changes to your code with confidence. Write unit tests with the intent to pass, and make changes to the code accordingly when they fail.

More on Best Practices:
https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices
https://confluence/x/cSyYAw
https://confluence/x/Lr5fAw

Writing Strong Tests

When writing tests, quality is much more important than quantity. We want to pick a suite of test cases small enough to run quickly, but large enough to validate the program.

We can choose our test cases by partitioning the input into subdomains, or similar inputs in which the program will have similar behavior.

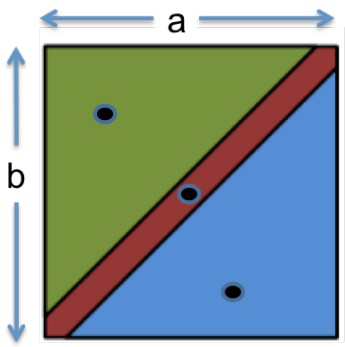
Example: C# Library Math.Max():

public static int Max (int a, int b)

Before moving to the next page, what are the ways to partition this method?

Answer:

- a < b
- a = b
- a > b



Still, there are ways we can improve our partitioning. Bugs often occur at the boundaries of these partitions, or "edge cases".

What would the edge cases be for the Max method?

Demo: Writing Testable Methods

Copy and paste this method into your IDE if you'd like to follow along

```
static void TimeOfDay()
    {
      DateTime tod = DateTime.Now;
      if (tod.Hour < 12)
      {
        Console.WriteLine("Good morning!");
      }
      else if (tod.Hour < 18)
      {
        Console.WriteLine("Good afternoon!");
      } else
      {
        Console.WriteLine("Good evening!");
      }
    }
```