Unit Testing Fundamentals

DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Agenda

- Why test?
- Unit testing principles
- Automated testing with MS Test
- Test Driven Development

Why Test?

- Catching bugs earlier is cheaper
  - Can be very expensive if bugs make it to production
- Confidence in code
  - Can prove it does what intended
- Refactor safely
  - Ensure that only structure has changed, not functionality

Different Kinds of Tests

- Testing can be performed at many levels
  - Unit testing
  - Integration testing
  - User acceptance testing
  - Load testing
  - UI testing
- All are important
  - Automate as much as possible

What is (or isn't) a Unit Test?

A test is not a unit test if:

Michael Feathers

- It talks to the database
- It communicates across the network
- It touches the file system
- It can't run at the same time as any of your other unit tests
- You have to do special things to your environment (such as editing config files) to run it.

Tests that do these things aren't bad. Often they are worth writing, and they can be written in a unit test harness. However, it is important to be able to separate them from true unit tests so that we can keep a set of tests that we can run fast whenever we make our changes.

Unit Test Principles

- Should only test a single unit (System Under Test)
  - Isolate from dependencies
- Should run quickly
  - Allows tests to be run frequently
- Should be independent
  - Tests can run with or without other tests and in any order
- Should have a meaningful name
  - Doesn't have to conform to normal naming conventions
  - Can be long as only called by testing framework

```
Credit_WhenPassedAnAmount_ShouldIncrementBal
                          anceByAmount
```

Unit Tests as Documentation

- Tests describe required functionality
  - Documentation of API
- Tests provide example usage
  - Allows other developers to see the code base in action

Common Arguments Against Unit Testing

- Our application is too complex
- I don’t want to have to maintain twice as much code
- Great in theory but I have a deadline
- This application is a short term tactical solution – we don’t need it

Structure of a Test - AAA

- Helpful to have a standard pattern for tests
  - Other developers know what to expect
- Arrange
  - Set up conditions for the test
- Act
  - Perform the actions under test
- Assert
  - Verify expected outcome was achieved

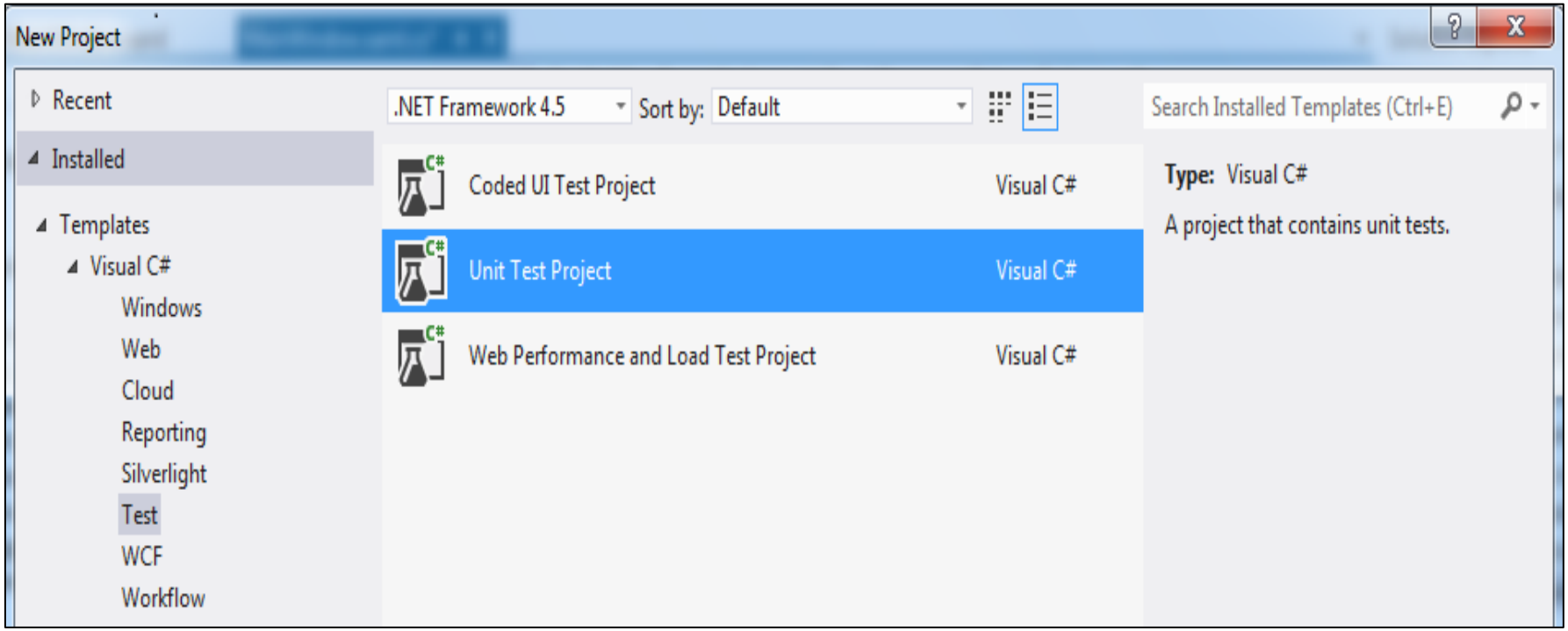
```
decimal initialBalance = 200m;
decimal creditAmount = 100m;
var acc = new Account(initialBalance);

acc.Credit(creditAmount);

Assert.AreEqual(initialBalance +
creditAmount, acc.Balance);
```

Unit Test Frameworks

- Automated test execution important
  - Allows regular running of the test suite
- A number of unit test frameworks for .NET
  - MS Test
  - NUnit
  - xUnit
  - TestDriven.NET
  - MbUnit
  - …
- Can be run within Visual Studio
  - Natively with VS 2012 and extensions
  - Via Resharper

MS Test

- Built into Visual Studio
  - Based on Nunit
- Create Test Project



Creating a Test

- Class annotated with [TestClass]
- Method annotated with [TestMethod]
  - Returns void
  - No parameters

```
[TestClass]
public class AccountTests
{
    [TestMethod]
    public void
Credit_WhenPassedAnAmount_ShouldIncrementBalanceByAmount()
    {
        // …
    }
}
```

Assertion

- Use Assert static class
  - AreEqual
  - IsNull
  - IsTrue
  - IsFalse
  - IsInstanceOf
  - …
- CollectionAssert for testing collections
  - E.g. has same members
- StringAssert for string specific
  - E.g. contains a substring

Testing for Exceptions

- Sometimes expected functionality is to throw an exception
  - Inputs incorrect
  - Processing fails
- Two models for dealing with exceptions
  - Use try / catch
  - Use built in [ExpectedException]

Using try/catch for Testing Exceptions

- Need to ensure that you catch the exception and mark the test as succeeded

```
[TestMethod]
public void
Ctor_WhenPassedNegativeInitialBalance_ShouldThrowExcept
ion()
{
    decimal initialBalance = -200m;

    try
    {
        var acc = new Account(initialBalance);
        Assert.Fail();
    }
    catch (ArgumentOutOfRangeException)
    {
        Assert.IsTrue(false);
    }
}
```

Using [ExpectedException] for Testing Exceptions

- [ExpectedException] can be more concise
  - No need for try/catch
- Can hide real cause of issue
  - E.g. Exception thrown in arrange phase

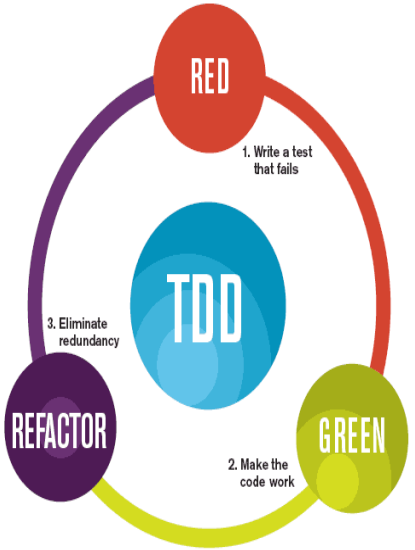
```
[TestMethod]
[ExpectedException(typeof(ArgumentOutOfRangeException))]
public void
Ctor_WhenPassedNegativeInitialBalance_ShouldThrowException(
)
{
    decimal initialBalance = -200m;

    var acc = new Account(initialBalance);
}
```

Test Driven Development

- Write tests before functionality
  - All functionality has an associated test
  - All functionality proven to work before moving on
  - Can refactor with little danger as high test coverage
- Simple to use with appropriate tooling
  - VS natively and Resharper can both generate code to ensure test compiles

Red / Green / Refactor

- Always write test to fail initially
  - Should generally be easy as haven't written code yet
- Write code to satisfy test
  - Write minimal code to pass test
- Refactor code to ensure good structure
  - Not required step but should always give yourself opportunity
  - Always rerun tests to ensure that noting has been broken



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

Benefits of TDD

- All functionality proven to work
- High code coverage
  - Not and end in itself but a useful indicator
- Iterative development can be proven not to break existing functionality
- Forces loosely coupled design
  - Must design to abstraction to be able to test

Summary

- Testing is a fact of life for developers
- Automated unit tests drive quality upwards
- Unit test framework built into Visual Studio
- TDD forces the issue and tends to produce better code