Introduction to Unit Testing

What is Unit Testing?

- Testing the behavior of software components apart from their actual use in a full software system
- Normally done with a “unit testing framework” which allows tests to be discoverable & run
- Creates a safety net around the code base to help identify side effects of changes
- It proves that our code still does what we intended it to do … the tests are effectively “executable specifications”

Not a Silver Bullet

- Does NOT show the absence of errors
- Does NOT catch integration errors
- The test code itself may contain errors

I SPENT THE WEEK WRITING A TEST SCRIPT FOR OUR PRODUCT.
AND I WROTE A TEST SCRIPT TO TEST DILBERT'S TEST SCRIPT.
YOUR SCRIPT WAS ALMOST PERFECT. KEEP UP THE GOOD WORK, BUDDY.

Dilbert.com DilbertCartoonist@gmail.com

3-24-11 ©2011 Scott Adams, Inc./Dist. by UFS, Inc.

Difficulties of Unit Testing Legacy Code

- Colossal Classes and Monstrous Methods
  (theoretically, imagine a class over 20,000 lines long with methods over 500 lines long with umpteen control paths)

- Hard Coupling
  (one class that relies on another class that might do something "heavy"; e.g., access a database, connect to another system, etc.)

- Overly Responsible Classes
  (classes that become dumping grounds for random responsibilities)

- Undefined Behavior
  (nobody knows what it really is supposed to do)

Why We Don't Write Unit Tests

I don't always test my code...
But When I Do, I Test In Production.

- Changing software development habits is hard work
- It takes time to code tests
- We'd rather be cranking out new functionality
- The investment value is not immediately obvious

Design for Testability

build components
that can be plugged together

- Rigorously follow the Single Responsibility Principle
- Use Dependencies via Interfaces
- Do not use the Singleton Pattern which prohibits replacement of the instance with a test object

NO!

What to Test

focus on behavior
… not implementation details

- Public interface of a class
- State transitions
- Calculations
- Polymorphism
- Operators

What to Test

Spend more time making tests for things that:

- Confirm business requirements
- Have a higher risk of changing
- Assist regression testing

Don’t waste time testing:

- Low risk items
  ```
  sut.name = “Bob”; Assert.IsEqual(“Bob”, sut.name);
  ```
- Code from somewhere else
  (.NET, Third Parties)

TDD
test driven design/development

- Writing unit tests before writing any real code:
  1. Write a test and run it (and it fails)
  2. Write code & run tests until it passes
  3. Refactor & run all tests until they all pass
  4. GOTO 1

- Or write tests immediately for all known behaviors; the failing tests are your TODO list.

- If you write real tests,
  you are forced to write testable code

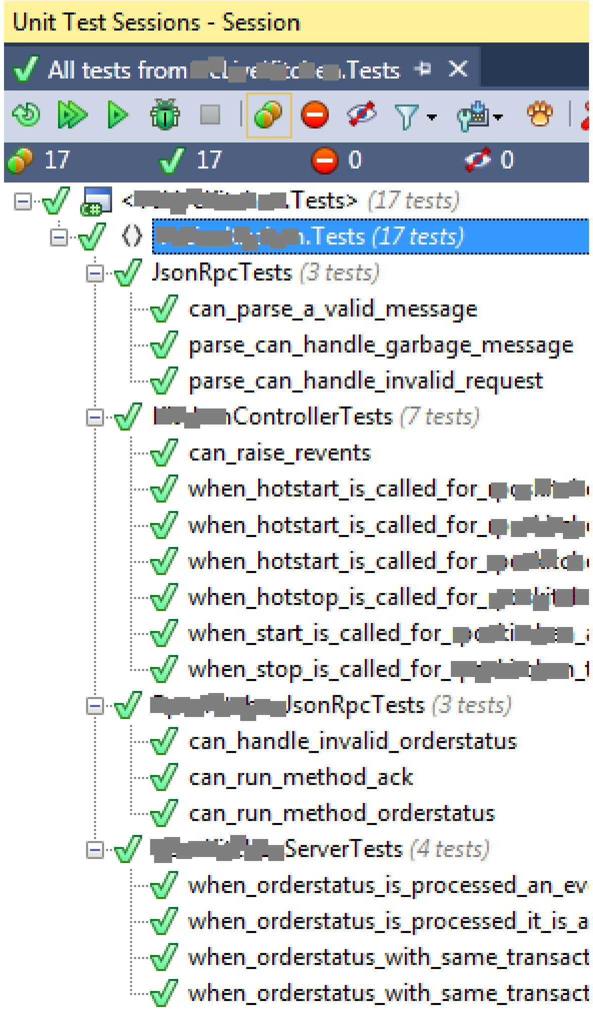
- “Play by Play: TDD with Brad Wilson” on PluralSight

Unit Testing Frameworks

The major frameworks for .NET:

- MSTest
  Microsoft’s built-in
- NUnit
  historically popular
- xUnit.net
  a rewrite by the original author of NUnit

They are all compatible with ReSharper and can be integrated into TFS Build



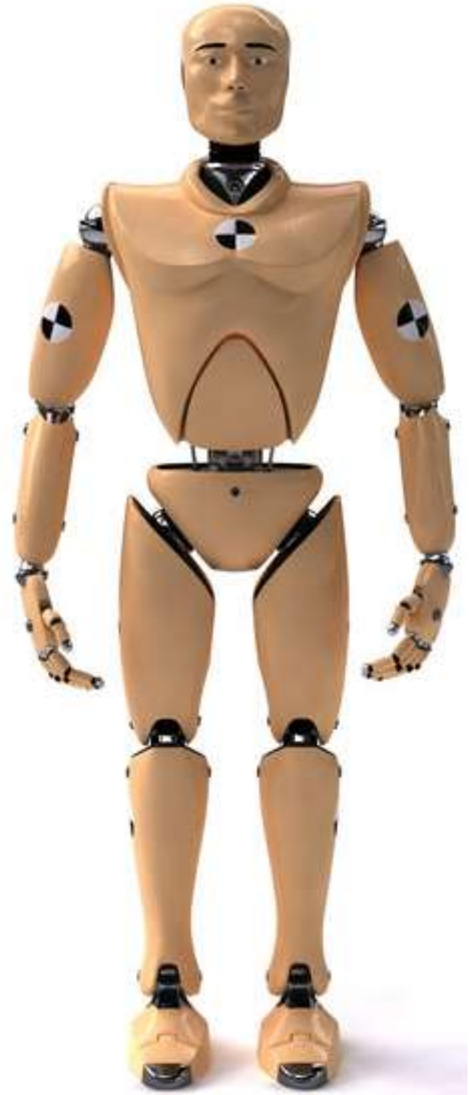
Unit Test Sessions - Session

✓ All tests from [redacted].Tests

17 ✓ 17 0 0

```
⊟ ✓ <[redacted]Tests> (17 tests)
  ⊟ ✓ {} [redacted].Tests (17 tests)
    ⊟ ✓ JsonRpcTests (3 tests)
        ✓ can_parse_a_valid_message
        ✓ parse_can_handle_garbage_message
        ✓ parse_can_handle_invalid_request
    ⊟ ✓ [redacted]ControllerTests (7 tests)
        ✓ can_raise_revents
        ✓ when_hotstart_is_called_for_[redacted]
        ✓ when_hotstart_is_called_for_[redacted]
        ✓ when_hotstart_is_called_for_[redacted]
        ✓ when_hotstop_is_called_for_[redacted]
        ✓ when_start_is_called_for_[redacted]
        ✓ when_stop_is_called_for_[redacted]
    ⊟ ✓ [redacted]sonRpcTests (3 tests)
        ✓ can_handle_invalid_orderstatus
        ✓ can_run_method_ack
        ✓ can_run_method_orderstatus
    ⊟ ✓ [redacted]ServerTests (4 tests)
        ✓ when_orderstatus_is_processed_an_ev
        ✓ when_orderstatus_is_processed_it_is_a
        ✓ when_orderstatus_with_same_transact
        ✓ when_orderstatus_with_same_transact
```

Mocking



Mimicking dependencies in a unit test:

- Test Dummy: fake data
- Stubs: implement an interface
- Fake: a stub that has been filled
- Spy: tracking call counts

Mocking libraries put this all together

- Rhino Mocks … last commit was Jan 2010
- Moq … actively maintained

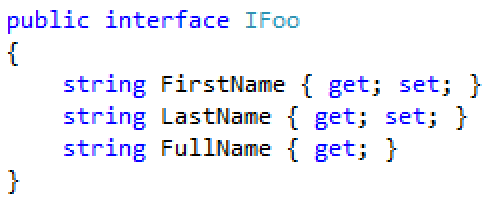
How to Write a Unit Test

Keep tests simple:

- Test a single behavior
- Self-contained (do own setup/teardown)
- Follow the structure: Arrange, Act, Assert

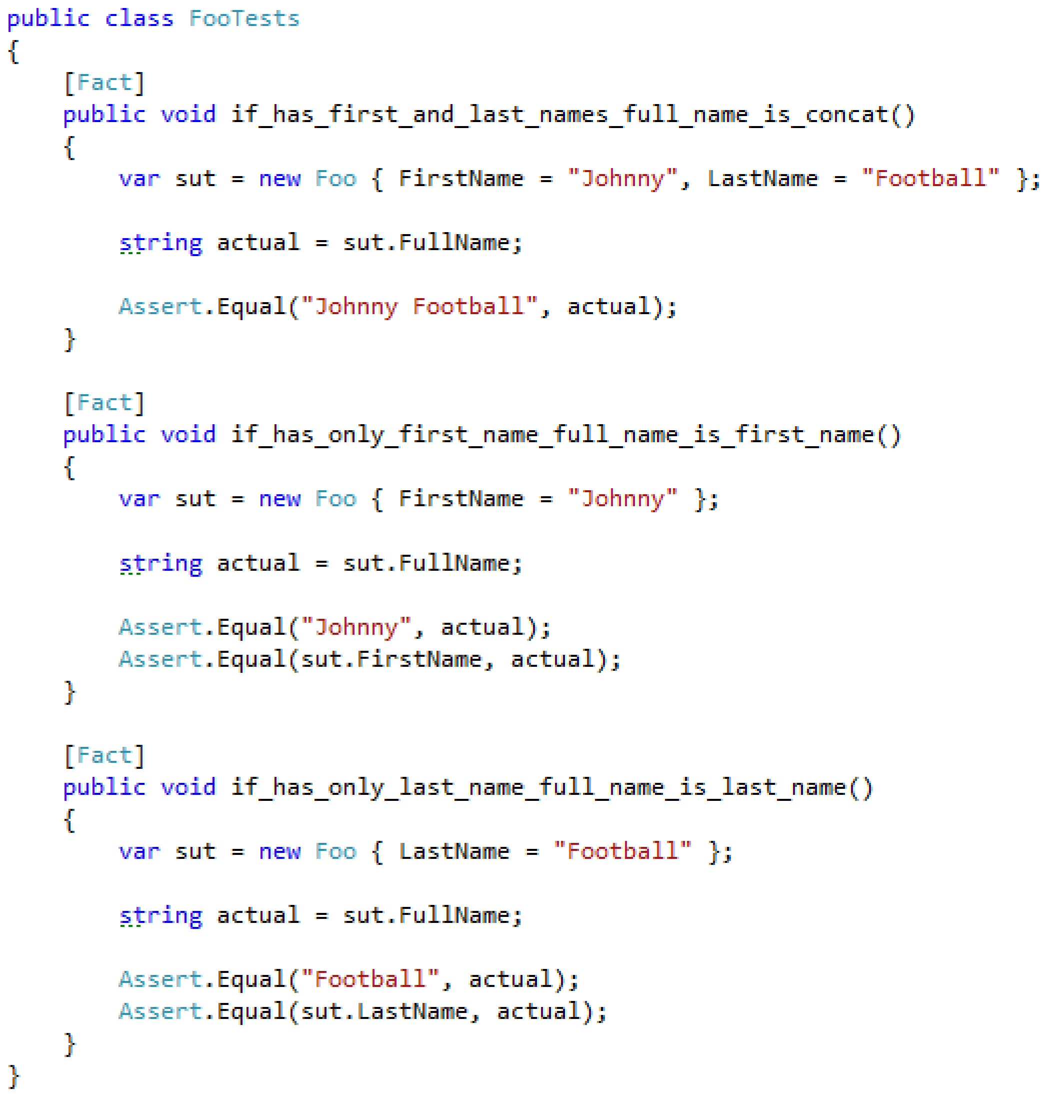
A Very Simple Example

Given a class that implements the following properties:



```
public interface IFoo
{
    string FirstName { get; set; }
    string LastName { get; set; }
    string FullName { get; }
}
```

The tests on the right check the expected behavior.

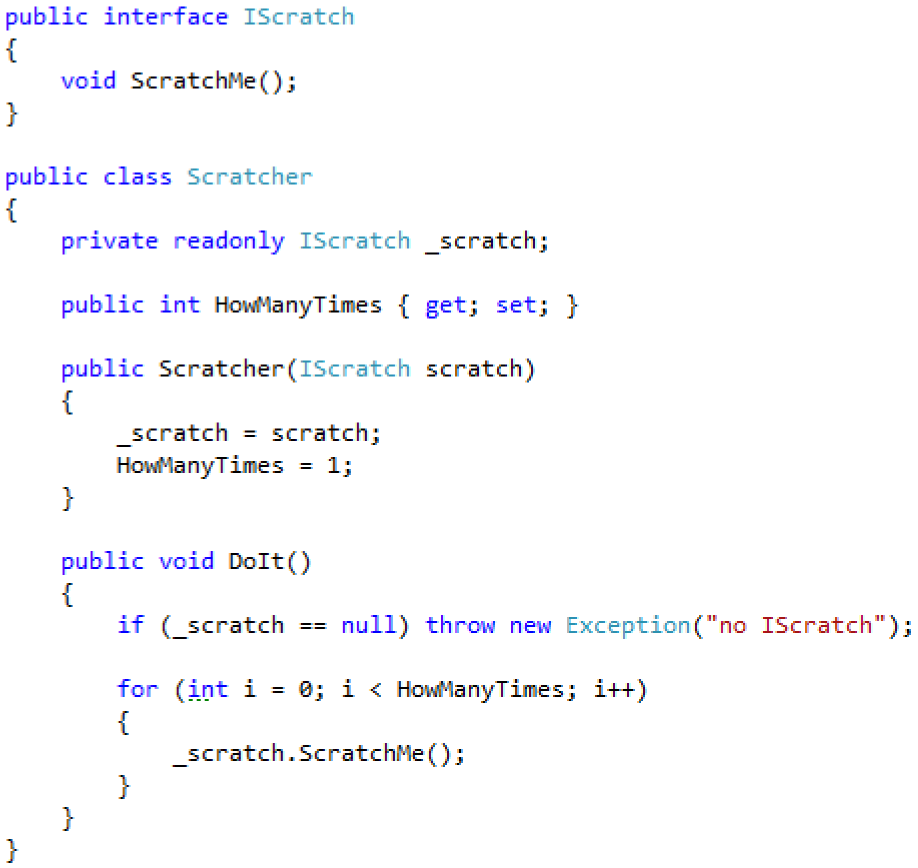


```
public class FooTests
{
    [Fact]
    public void if_has_first_and_last_names_full_name_is_concat()
    {
        var sut = new Foo { FirstName = "Johnny", LastName = "Football" };

        string actual = sut.FullName;

        Assert.Equal("Johnny Football", actual);
    }

    [Fact]
    public void if_has_only_first_name_full_name_is_first_name()
    {
        var sut = new Foo { FirstName = "Johnny" };

        string actual = sut.FullName;

        Assert.Equal("Johnny", actual);
        Assert.Equal(sut.FirstName, actual);
    }

    [Fact]
    public void if_has_only_last_name_full_name_is_last_name()
    {
        var sut = new Foo { LastName = "Football" };

        string actual = sut.FullName;

        Assert.Equal("Football", actual);
        Assert.Equal(sut.LastName, actual);
    }
}
```

A Simple Mock Example

an IScratch can ScratchMe()

a Scratcher uses an IScratch to ScratchMe()

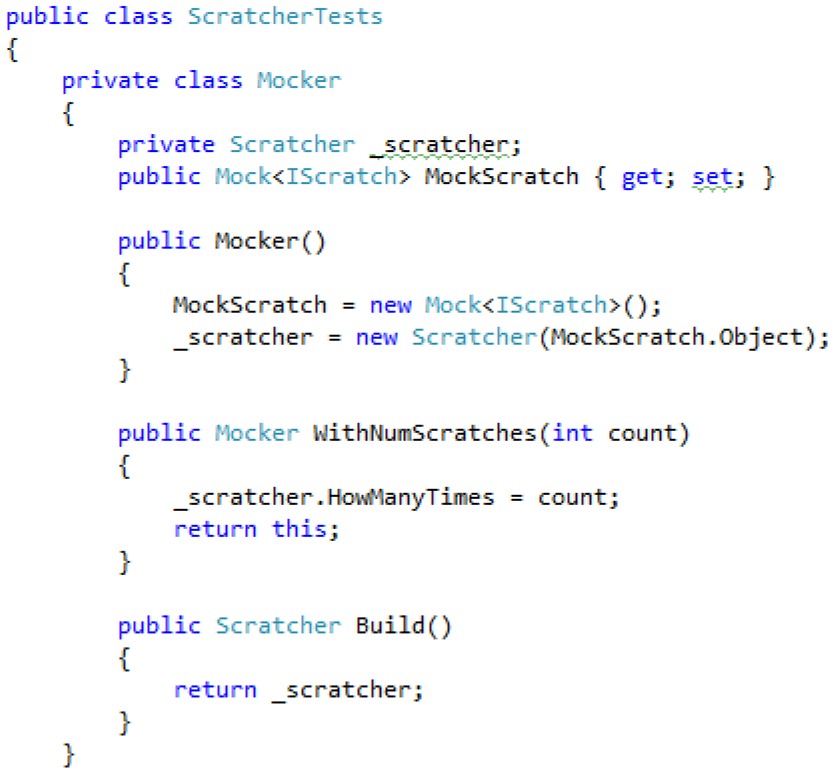


```
public interface IScratch
{
    void ScratchMe();
}

public class Scratcher
{
    private readonly IScratch _scratch;

    public int HowManyTimes { get; set; }

    public Scratcher(IScratch scratch)
    {
        _scratch = scratch;
        HowManyTimes = 1;
    }

    public void DoIt()
    {
        if (_scratch == null) throw new Exception("no IScratch");

        for (int i = 0; i < HowManyTimes; i++)
        {
            _scratch.ScratchMe();
        }
    }
}
```

Encapsulate SUT Configuration

The private Mocker class uses the fluent builder pattern.

It encapsulates construction and configuration of all mocks and the system under test.

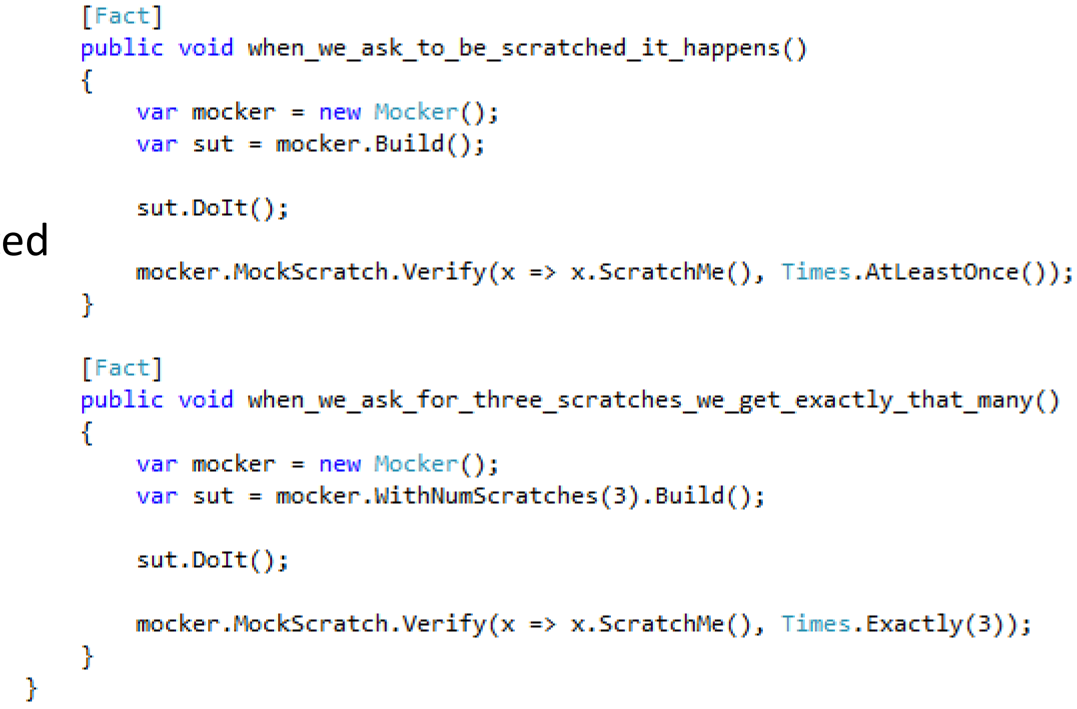


```
public class ScratcherTests
{
    private class Mocker
    {
        private Scratcher _scratcher;
        public Mock<IScratch> MockScratch { get; set; }

        public Mocker()
        {
            MockScratch = new Mock<IScratch>();
            _scratcher = new Scratcher(MockScratch.Object);
        }

        public Mocker WithNumScratches(int count)
        {
            _scratcher.HowManyTimes = count;
            return this;
        }

        public Scratcher Build()
        {
            return _scratcher;
        }
    }
```

Arrange, Act, Assert

1. Arrange a default system
2. Act on the system
3. Assert that it behaved as expected

This is very similar to the first test but does further configuration of the system under test.



```
    [Fact]
    public void when_we_ask_to_be_scratched_it_happens()
    {
        var mocker = new Mocker();
        var sut = mocker.Build();

        sut.DoIt();

        mocker.MockScratch.Verify(x => x.ScratchMe(), Times.AtLeastOnce());
    }

    [Fact]
    public void when_we_ask_for_three_scratches_we_get_exactly_that_many()
    {
        var mocker = new Mocker();
        var sut = mocker.WithNumScratches(3).Build();

        sut.DoIt();

        mocker.MockScratch.Verify(x => x.ScratchMe(), Times.Exactly(3));
    }
}
```