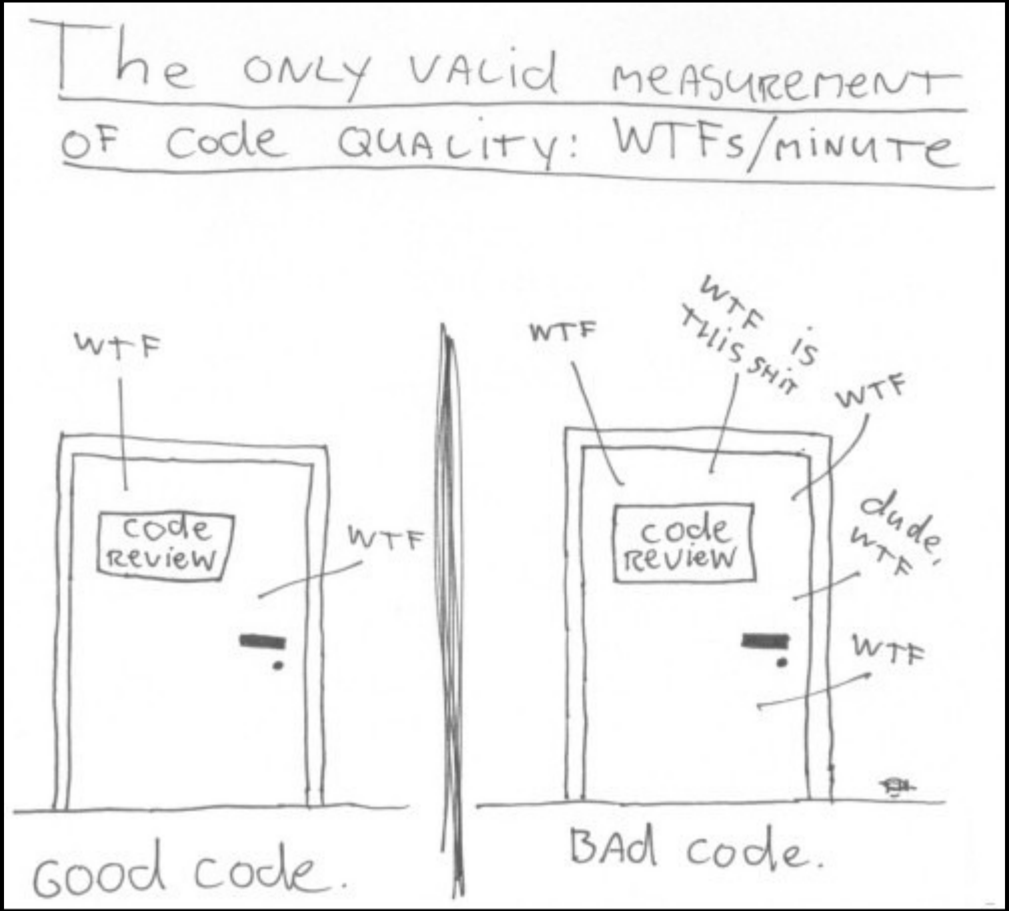
Unit Testing



(c) 2008 Focus Shift/OSNews/Thom Holwerda - http://www.osnews.com/comics

Part One

Introduction

Exercise

What is a Unit Test ?

What is a Unit Test ?

“A **Test method** which verifies the correct behaviour of a single **Unit of Work**.”

- Fast-running
- Trustworthy

Exercise

Why Unit Test ?

Why Unit Test ?

Reason One: To test your code…

- New code (developer acceptance test)
- Updated code (regression test)

Why Unit Test ?

Reason One: To test your code…

- New code (developer acceptance test)
- Updated code (regression test)

TDD’ers will tell you this is not the reason they write Unit Tests

Why Unit Test ?

Reason Two: To verify / flesh-out requirements

= Tests as Specification

Why Unit Test ?

Reason Three: To better understand a system

= Tests as Documentation

Why Unit Test ?

Reason Three: To better understand a system

= Tests as Documentation

Unit Tests tells you **what** the system should do.
Code tells you **how** it is done.

Why Unit Test ?

Reason Four: To reduce risk

- Refactoring

Why Unit Test ?

Reason Five: To write better architected code

- **Focus** efforts on only writing enough code to make a failing test case pass
- Approach development from the **Client perspective** (rather than the Server)

Why Unit Test ?

Reason Six: To save time

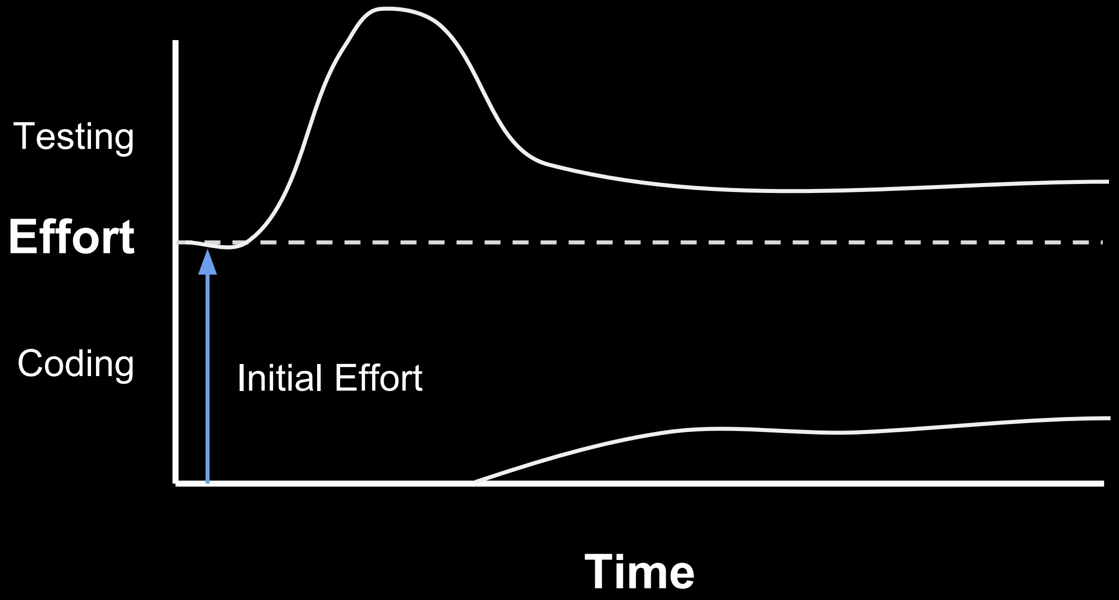
Modest up-front investment

Pays dividends later on

Diagram

Cost Benefit over Time

Cost Benefit over Time



How to do Unit Testing

~~7~~ 8 Habits of Highly Effective Tests

1. Simple
2. Fast-running
3. Single execution path
4. Verify a single condition
5. Developed as 1st-class citizens
6. Intent-revealing coding
7. Write your test first (TDD)
8. Isolate your SUT

Terms

SUT = System Under Test

DOC = Depended-On Component

TCC = Test Case Class

Fixture = execution context for a test (created by TCC)

4 Phase Test

xUnit / TDD / BDD

Setup / Arrange / Given

Execute / Act / When

Verify / Assert / Then

Tear-down

Demo

xUnit Refactoring Walkthrough

Part Two

Naming Conventions

Exercise

Unit Test Naming

Feature: Flight Management

Scenario: Cancelling a Flight

Given a Flight
When I cancel the Flight
Then the Flight Status should be Cancelled

Feature: Online Payments

Scenario: making a Payment from an Account with Insufficient Funds.

Given an Account with no Available Balance
When I try to make a Payment
Then the Payment should be Rejected

Feature: User trades stocks

Scenario: User requests a Sell before Close of Trading

Given I have 100 shares of MSFT stock
And I have 150 shares of APPL stock
And the time is before close of trading
When I ask to sell 20 shares of MSFT stock
Then I should have 80 shares of MSFT stock
And I should have 150 shares of APPL stock
And a sell order for 20 shares of MSFT stock should have been executed

Unit Test Naming Goals

- Unambiguous
- Concise
- Use of Ubiquitous Language
- Consistent

Unambiguous Name

Components of a Unit Test name:

- Method
- Scenario:
  - Initial state of SUT
  - Key Inputs
- Expected Outcome

Test Naming Strategies

- ‘Freestyle’
- Structured

Freestyle Test Naming

- TestNameInPascalNotation

- Or_use_snake_notation_for_readability

Freestyle

Divide_by_zero_should_throw_exception

If_no_accounts_selected_should_retrieve_transactions_for_first_transactional_account

Adding_3_integers_should_return_sum

Freestyle

Divide_by_zero_should_throw_exception

If_no_accounts_selected_should_retrieve_transactions_for_first_transactional_account

Adding_3_integers_should_return_sum

Freestyle

Should_throw_exception_when_divide_by_zero

Should_retrieve_transactions_for_first_transactional_account_if_no_accounts_selected

Should_return_sum_when_when_adding_3_integers

Structured Naming

- Method_Scenario_Outcome

- UnitOfWork_Scenaio_Outcome

Structured Naming

Divide_ByZero_ThrowsException

OnLoad_NoAccountsSelected_GetsTransactionsForFirstTransactionalAccount

Add_3Integers_ReturnsSum

Test Organisation & Naming

Organisation pattern:

Testcase Class Per Method

Alternative

- BDD Naming

User Story

Title: [some activity]

Narrative:

As a [role]
I want [feature]
So that [benefit]

Acceptance Criteria:

Scenario 1: [title]

Given [context]
When [event]
Then [outcome]

Scenario 2: [title]

Given [context]
When [event]
Then [outcome]

NSpec (1st Generation BDD)

```
void Given_10()
{
    before = () => calc = new Calculator();

    it["should return 11 when adding 1"]
    = () => calc.Add(10, 1).should_be(11);

    it["should throw divide by zero exception when dividing by 0"]
    = expect<DivideByZeroException>(() => calc.Divide(10, 0));
}
```

Demo

NSpec (1st Generation BDD)

```
void Given_10()
{
    before = () => calc = new Calculator();

    it["returns 11 when adding 1"]
    = () => calc.Add(10, 1).should_be(11);

    it["throws divide by zero exception when dividing by 0"]
    = expect<DivideByZeroException>(() => calc.Divide(10, 0));
}
```

SpecFlow (2nd Generation BDD)

Acceptance Criteria:

Scenario: Cancelling a Flight

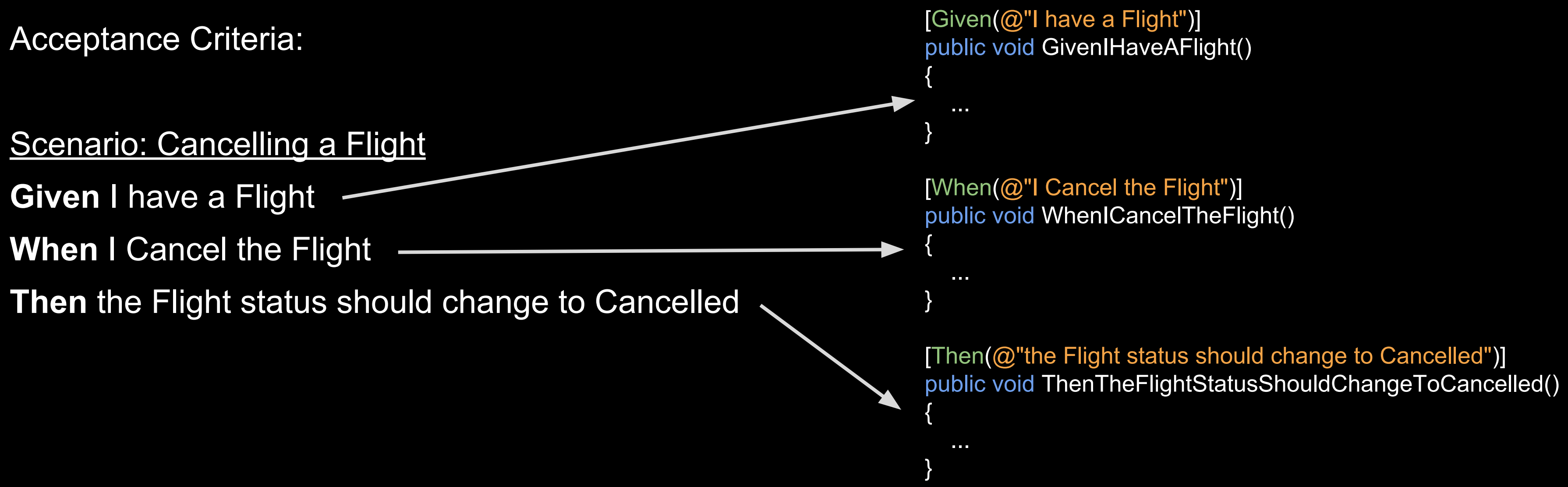
Given I have a Flight

When I Cancel the Flight

Then the Flight status should change to Cancelled

Demo

SpecFlow (2nd Generation BDD)



Acceptance Criteria:

Scenario: Cancelling a Flight

Given I have a Flight

When I Cancel the Flight

Then the Flight status should change to Cancelled

```
[Given(@"I have a Flight")]
public void GivenIHaveAFlight()
{
    ...
}

[When(@"I Cancel the Flight")]
public void WhenICancelTheFlight()
{
    ...
}

[Then(@"the Flight status should change to Cancelled")]
public void ThenTheFlightStatusShouldChangeToCancelled()
{
    ...
}
```

Part Three

Patterns

xUnit Patterns

4 Phase Test:

- Setup (Arrange)
- Execution (Act)
- Verification (Assert)
- Tear-down

Fixture Patterns

Life-cycle:

1. Fresh Fixture
    - New fixture for every test method

2. Shared Fixture
    - All test methods share common fixture

Demo

Setup (Arrange)

3 main setup patterns:

- Inline setup
- Delegated setup
- Implicit setup

Verification (Assert)

Two main verification strategies:

- State verification
- Behaviour verification

Diagram

State Verification

- Simpler (normal) approach
- Verify direct outputs of the SUT

Demo

State Verification

- Two variations:
    - Procedural state verification
    - Expected state specification

Behaviour Verification

- More complicated
- Dynamic - need to catch the SUT ‘in the act’
- Verify indirect outputs to DOCs
- Requires the use of specialised Test Doubles
- Set expectations on DOC methods

Test Doubles

- The stunt doubles of the unit testing world
- Flavours:
  - Stubs
  - Mocks
  - Spies
  - Fakes
  - Dummies

Isolation Frameworks

Isolate the SUT from DOCs

- Stubs
- Mocks
- Spies

Stubs

- Return ‘canned’ data
- Supply inputs into the SUT from DOC
- Can have multiple stubs in a unit test

Mocks

- Test **outputs** from the SUT
- Do not return data
- Verify that invocation expectations have been met
- There should only be a **single Mock per test**

Spies

- Specialised Stubs
- Return data *and* audit calls
- Only a single Spy or Mock per test

Creation Methods

- Extract common / complicated / irrelevant Setup logic into dedicated method

Creation Methods

- Allow intent-revealing coding
- Two variations:
  - Anonymous Creation Method
  - Parameterised Creation Method

- Test Code Duplication
- Obscure Test
  - Irrelevant Information

Custom Assertions

- Extract common / multi-step assertion logic into dedicated method

Custom Assertions

- Allow intent-revealing coding

- Test Code Duplication
- Obscure Test
  - Irrelevant Information

Part Four

Smells

Obscure Test

- A test which is difficult to understand
- Two main causes:
  - Too little information
  - Too much information

Too Little Information

- Mystery Guest
    - part of the Setup / Verification logic done outside the test.

Mystery Guest

```
public void testGetFlightsByFromAirport_OneOutboundFlight_mg()
{
        loadAirportsAndFlightsFromFile("test-flights.csv");

        // Exercise System
        List flightsAtOrigin = facade.getFlightsByOriginAirportCode( "YYC");

        // Verify Outcome
        assertEquals( 1, flightsAtOrigin.size());
        FlightDto firstFlight = (FlightDto) flightsAtOrigin.get(0);
        assertEquals( "Calgary", firstFlight.getOriginCity());
}
```

Too Much Information

- Eager Test
  - Trying to verify too many conditions in a single test
  - Can lead to Assertion Roulette
- Irrelevant Information
  - Inclusion of logic which doesn’t materially affect the test

Eager Test

```
public void testFlightMileage_asKm2() {
    // set up fixture
    // exercise constructor
    Flight newFlight = new Flight(validFlightNumber);
    // verify constructed object
    assertEquals(validFlightNumber, newFlight.number);
    assertEquals("", newFlight.airlineCode);
    assertNull(newFlight.airline);
    // set up mileage
    newFlight.setMileage(1122);
    // exercise mileage translator
    int actualKilometres = newFlight.getMileageAsKm();
    // verify results
    int expectedKilometres = 1810;
    assertEquals( expectedKilometres, actualKilometres);
    // now try it with a canceled flight
    newFlight.cancel();
    try {
        newFlight.getMileageAsKm();
        fail("Expected exception");
    } catch (InvalidRequestException e) {
        assertEquals( "Cannot get cancelled flight mileage",
        e.getMessage());
    }
}
```

Conditional Test Logic

- Test contains code that may or may not be executed

```
if(flightsFromCalgary != null) {
        i = flightsFromCalgary.iterator();
        while (i.hasNext()) {
                FlightDto flightDto = (FlightDto) i.next();
                if (flightDto.getFlightNumber().equals(
                expectedCalgaryToVan.getFlightNumber()))
                {
                        assertEquals("Flight from Calgary to Vancouver",
                                expectedCalgaryToVan,
                                flightDto);
                        break;
                }
        }
}
```

```
flightsFromCalgary.Should().Not.BeNull();   // guard assertion

i = flightsFromCalgary.iterator();
while (i.hasNext()) {
    FlightDto flightDto = (FlightDto) i.next();
    if (flightDto.getFlightNumber().equals(
    expectedCalgaryToVan.getFlightNumber()))
    {
        assertEquals("Flight from Calgary to Vancouver",
            expectedCalgaryToVan,
            flightDto);
        break;
    }
}
```

```
flightsFromCalgary.Should().Not.BeNull();   // guard assertion

flightDto = FindFlight(flightsFromCalgary, expectedCalgaryToVan);  // test utility method

assertEquals("Flight from Calgary to Vancouver",
        expectedCalgaryToVan,
        flightDto);
```

Test Code Duplication

- The same test code is repeated many times

Part Five

Tooling

Unit Testing Frameworks

- Framework
  - Attributes
  - Assertion methods

- Test Runner
  - Executing tests
  - Displaying results

Test Runners

- Visual Studio
    - Test Runner
        - Built-in support for MSTest
        - Plugins for other frameworks
    - 3rd Party - Resharper, TestDriven.NET
    - Command-Line
- Custom
    - NUnit test runner
    - XUnit GUI

MSTest

- Integrated into Visual Studio
- Basic
- Not extensible
- No parameterised Tests

NUnit

- Proven - but dated
- Uses stand-alone test runner
- Can use plugin to integrate with VS

XUnit

- NUnit modernised
- Good extensibility
- Opinionated
- Assert.Throws()

Part Six

TDD Kata

String Calculator

```
int Add(string numbers)
```

Requirements

1. An empty string returns 0
2. A single number return the value
3. Two numbers, comma delimited, returns the sum
4. Two numbers, newline delimited, returns the sum
5. Three numbers, delimited either way, returns the sum
6. Negative numbers throw an exception
7. Numbers greater than 1000 are ignored
8. A single char delimiter can be defined on the first line (e.g. //# for a ‘#’ as the delimiter)
9. A multi char delimiter can be defined on the first line (e.g. //[###] for ‘###’ as the delimiter)