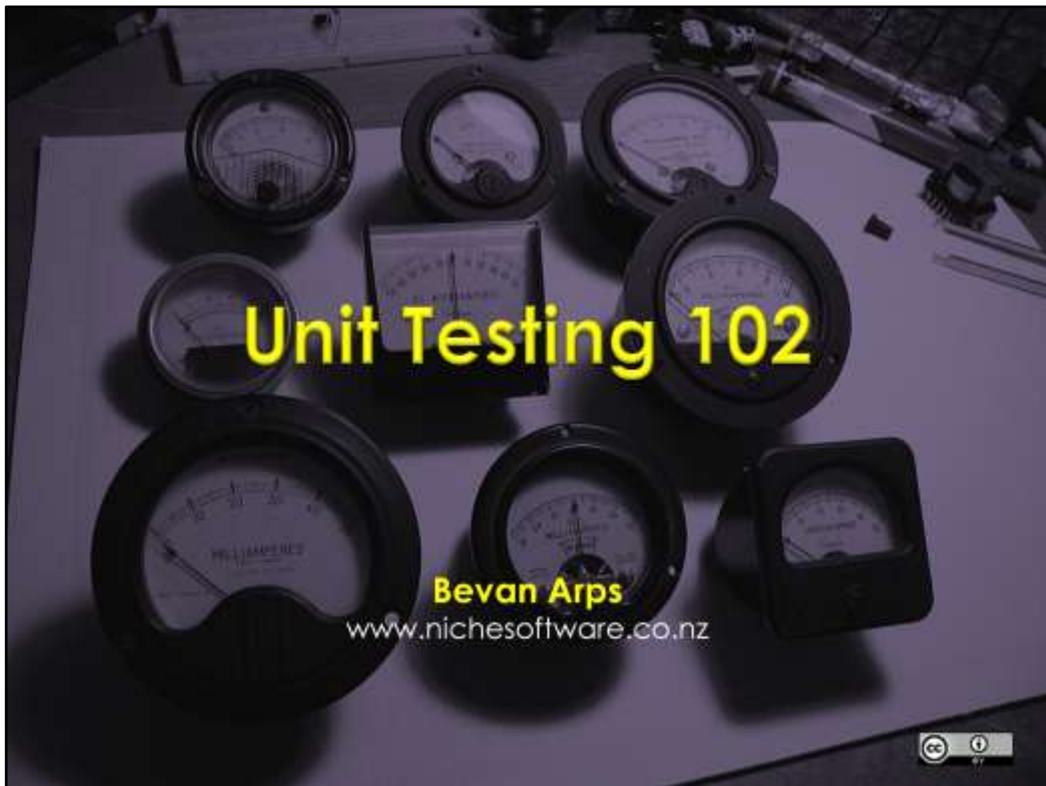
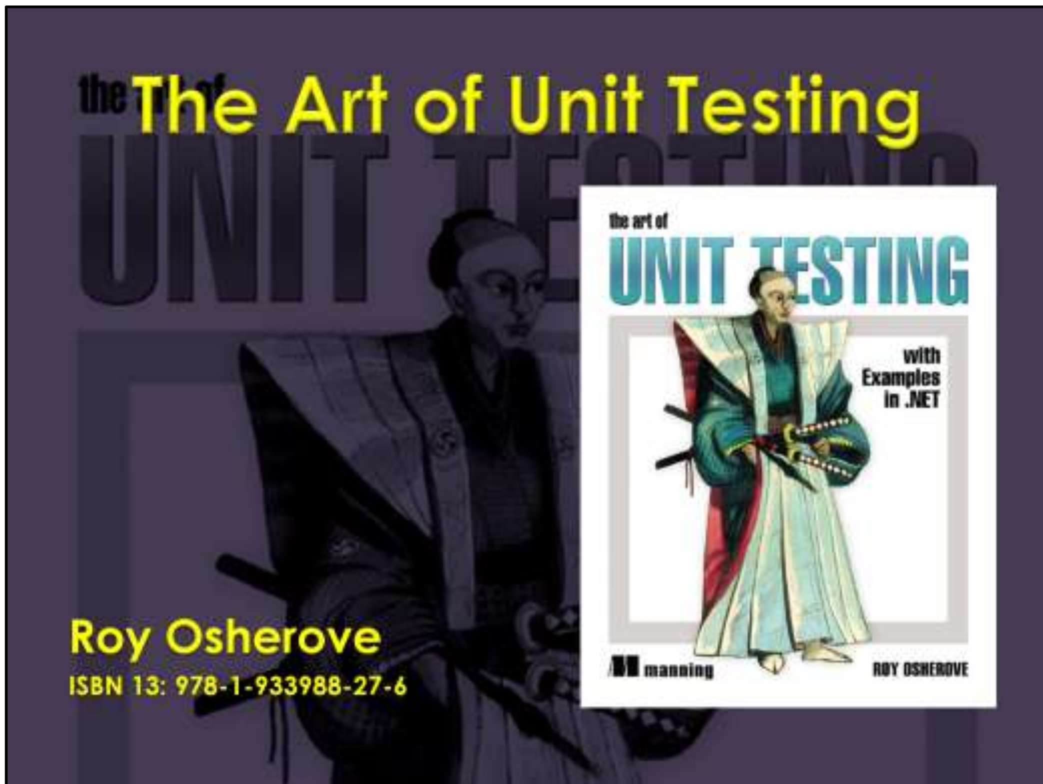


Who am I?

I’ve been programming for 27 years
First computer was a Sinclair ZX81 with a 16K ram pack
Parents refused to buy me games, so I set out to write my own ...
... it was all downhill from there.

This is a modified form of a presentation delivered to the Wellington .NET User Group in November 2009.



Key reference used for this presentation

I've been trying to get my head around unit testing for years
(My earliest bog entry on this was in October 2004)

While I've been using unit testing on and off, with some success
this was the book that really made things click,
tying together the bits I knew with some more into a coherent whole

What is a unit Test?

Don't get too hung up on the definition of "unit".
Could be a single method – a single feature – a single interaction

I've seen these called micro-tests instead.

A Unit test runs ...
... on demand ... without user interaction ... quickly
... entirely in memory ... without any external dependencies

For this presentation
Using .NET - C#, 3.5
NUnit, version 2.5,
Assert.That() syntax

Separate assembly for unit tests
Name for the assembly being tested, with suffix .Tests
Integration tests also in separate assembly, suffix .IntegrationTests

At least one test class per class in your application

Naming is important

Demo: Show structure in sample solution

Three part naming system – from Roy Oscherove

Based on method naming, often the only thing included in the results shown by test runners

The method under test
Groups all tests for the one method together

The scenario being tested
Groups all tests for the same scenario together

The expected outcome
This is the thing we assert

Demonstration

Show some existing test names.
Review RequireTests – and make them pass

Review RuleTests – and make them pass too

Our applications are comprised of a large number of discrete pieces that we try to test in isolation
But sometimes it can be difficult to isolate individual pieces

Ways to isolate things
Override Methods
Service Locator Pattern
Dependency Injection

Last two involve using "stunt doubles" that look like the real thing but aren't
But, not getting into code mocking frameworks – that's another topic for another time

Static Methods
Dependencies on Static classes and methods can be a problem
Solution: isolate within virtual methods – override to create isolated fake
Example: Person.SetLoyaltyProgram()
Write tests and show how to isolate dependency on Static Class LoyaltyProgramRepository
[Factor out method FindLoyaltyProgramByCode()]

Service Locator
Dependencies on particular implementations can be a problem
Solution: Isolate behind an interface and use Service Locator to obtain the instance
Example: Person.SetLoyaltyProgram()
Write tests and show how to isolate dependency on EventLogger

Dependency Injection
Another way to resolve a dependency on a particular implementation
Don't create your own instance, but instead rely on an instance provided to you from elsewhere
Example: Create Person.SetLoyaltyProgram(ILogger)
Possibly omit for time.

What makes a good test?

Trustworthy – that when they work or fail you believe the result
Readable – that when you read the code it makes sense
Maintainable – that you find it easy to update the test when required

Test code is still code – needs to be maintainable, held to high standards

Useful techniques
Use Setup() for stuff that is common to ALL tests
Private Factory methods

A good test has exactly one Assert

Why?

Many (most, if not all) test suites signal a test failure by throwing an exception

If a test contains multiple asserts,
An early failure conceals later results.

Limiting to ONE assert also helps to limit complexity
Keeping the tests short and simple.

Test Driven Development

Write the test *before* you write the code.

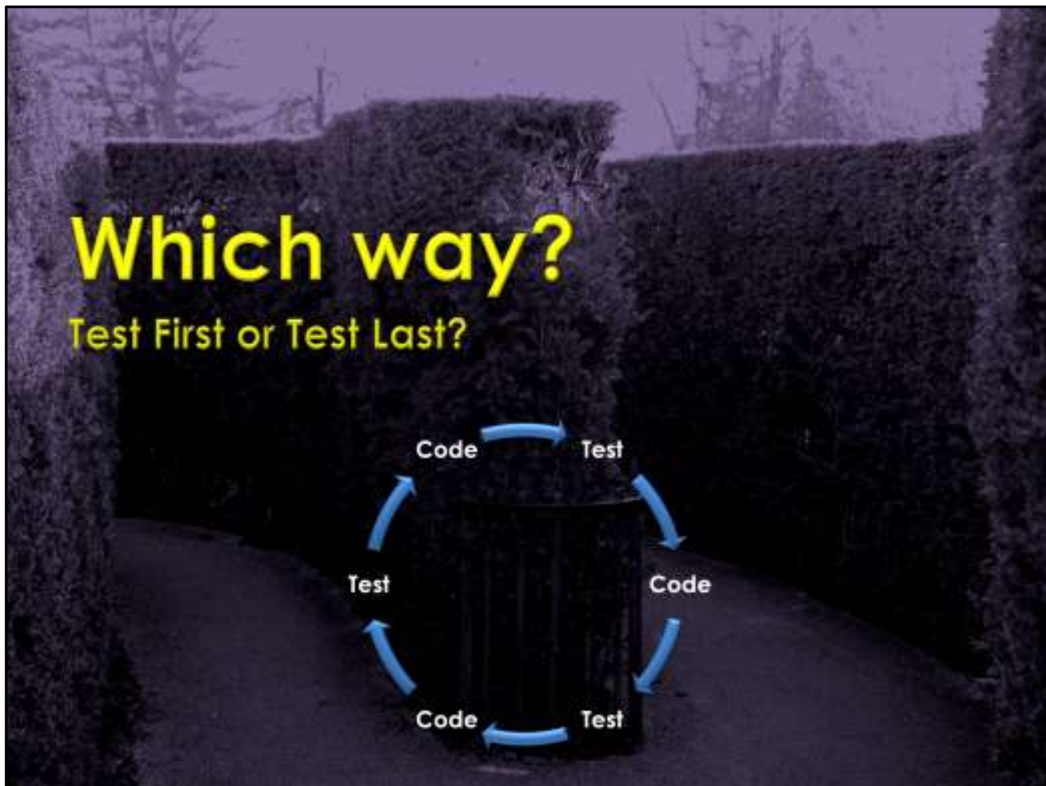
Red – write a test that fails. Often, it won't even compile.
Green – write the simplest reasonable code to make the test pass
Refactor – tidy up your code to make it sound

Reinforce previous comment: Test code is still code. Treat it well.

Show how Test Driven Development can actually work

Empty ValidationResult class
Write each test in ValidationResultTests, make it pass

Note that we only concentrate on tests of ValidationResult
Tests for ValidationSet are elsewhere



So, which way to go?

Write the test *before* the code
or
Write the test *after* the code

Test First gives you the opportunity to flesh out what your code should do, before you write it.
This design process will influence your design – mostly in good ways

Test Last allows you to continue working the way that you have

Mostly though, you should be writing a little bit of each.

Key observation: Write tests. Anytime.

Any questions kept until the end?

Bevan Arps
bevan@nichesoftware.co.nz
http://www.nichesoftware.co.nz