Effective Unit Testing

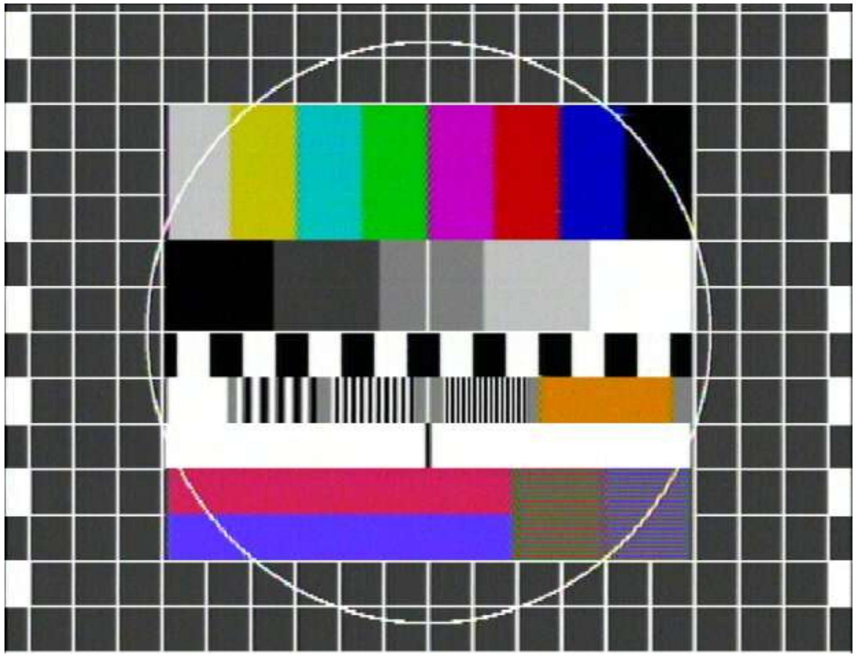
Brad Wilson
Scott Densmore

Agenda

- What is Unit Testing?
- Typical Unit Testing Problems
- Best Practices for Effective Unit Testing

What is Unit Testing?

A Definition

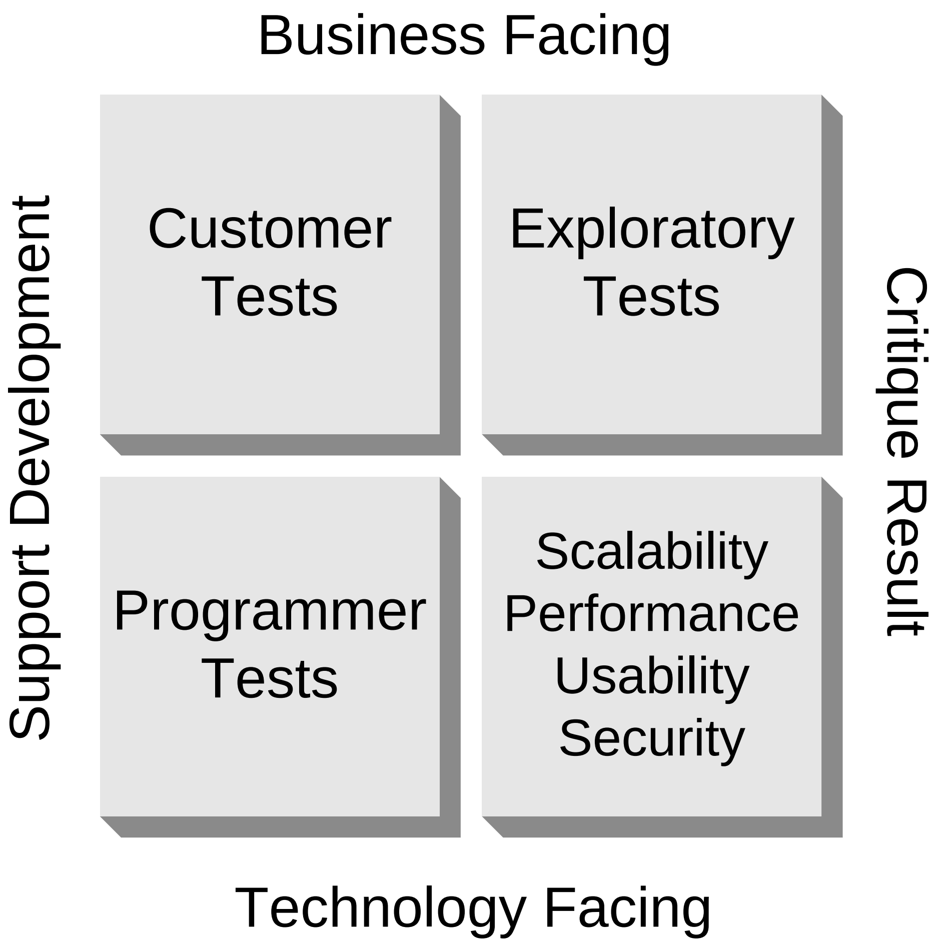


- Unit Testing is code that…
  - Is written by developers, for developers.
  - Exercises a small, specific area of functionality.
  - Helps “prove” that a piece of code does what the developer expects it to do.

What Unit Testing Is Not

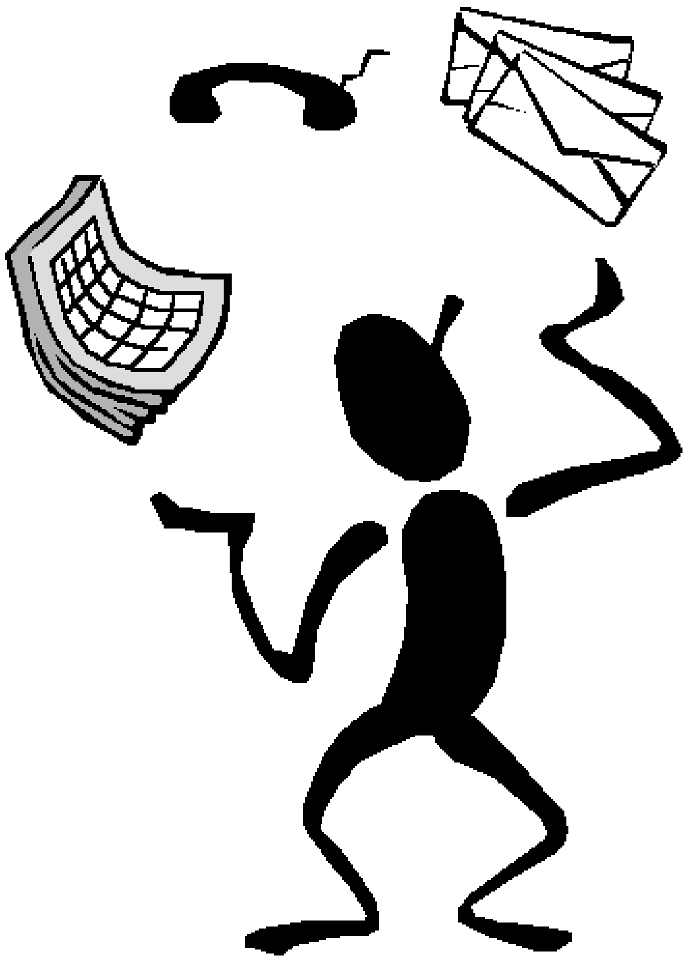
- Acceptance Testing (aka Functional Testing)
- Performance Testing
- Scalability Testing

Marick's Four Quadrants of Testing



Source: Brian Marick – http://testing.com/

Why Unit Test



- It will make your life easier
- Better code
- Better designs
- Code is easier to maintain later
- Confidence when you code

Common Excuses

- I'm not a tester!
- It takes too much time.
- It takes too long to run the tests.
- I don't know how to test it.
- I don't really know what it is supposed to do, so I can't test it.
- But it compiles! It doesn't need tests.

Typical Unit Testing Problems

Testing Is Monotonous and Boring

- Often indicates bad tooling or bad techniques
- Unit testing should…
  - Be easy to do
  - Provide rapid feedback

Poor Test Coverage

- Often happens when tests not written first
- Hard to retrofit later
- Test coverage is orthogonal to test quality

Purpose of Tests is Misunderstood

- Remember why we do unit testing
- Easy to get distracted by irrelevant things
- What is the scope of the test?

Informal Testing Process

- How does your team do unit testing?
  - Test-first?
  - Intuitive “poking and prodding” style testing?
  - Smoke testing?
  - Whatever the developer wants?

Inconsistent Testing

- Are tests required before check in?
- How is it enforced?

Low Test Quality

- Single biggest problem in unit testing today
- Testing requires a special mindset
- Tools only solve half the problem

Code Not Designed for Test

- Often the cause of low test quality
- Hard to design testable code
- However, testable code often is better designed

Tests Not Maintained

- So you’ve shipped your code and someone finds a bug…
- Do you fix the tests? Or just fix the bug?
- What about the rest of the team?

Best Practices for Effective Unit Testing

Automate Your Tests

- If it is hard to run tests, you won’t do it
- Manual (aka scripted) tests make regression testing very hard

Use Good Tools

- The tools should make it…
  - Easy to write tests
  - Easy to organize your tests
  - Easy to run tests (all, some, one)
- They should provide…
  - Quick feedback of pass/fail
  - Details on the fail conditions

Use Good Tools

<u>Testing Focused Tools</u>

- NUnit and/or MSTest
- Test Driven .NET
- NMock

<u>Supporting Tools</u>

- MSBuild
- Resharper
- CodeRush
- CodeSmith

Get a Mentor

- Testing is as much an art as a science
- Learning to write good tests is hard
- Some people are naturally good at it and some aren’t

Use a Test List

- Start your development activities by writing down a list of things you want to test
- You will often think of a test while writing another one. When you do, add it to the list.
- Review your list frequently

Write Tests First

- Test-driven development (TDD) is a proven way of improving quality*
- TDD’s main objective is not testing software! (this is a side effect)
- TDD’s main objective is to aid programmers and customers during the development process with unambiguous requirements.
- Code is written with testing as a primary motivation. In short, the code is testable.

* http://collaboration.csc.ncsu.edu/laurie/Papers/TDDpaperv8.pdf

What to Test

- The “did we get what we expected” test isn’t always good enough

- Ask yourself the question…
  - “If the code ran correctly, how would I know?”

What to Test - Testing Heuristics

- Test at the boundaries
- Test every error message
- Test different configurations
- Run tests that are annoying to setup
- Avoid redundant tests
  - Source: Lessons Learned in Software Testing, Cem Kaner, James Bach, Brett Pettichord

What to Test – BICEP

B	Are the boundary conditions correct?
I	Can you check inverse relationships?
C	Can you cross check using other means?
E	Can you force an error condition to happen?
P	Are the performance characteristics acceptable?

Source:
Pragmatic Unit Testing in C# with NUnit, Andy Hunt and Dave Thomas

Test Structure

- William Wake’s 3-A Pattern
  - Arrange
  - Act
  - Assert

Designing for Test

- One simple question to help you write good code…

“How am I going to test this?”

- If you don’t know the answer, then you probably need to reconsider your design.

What is a Mock Object?

"A mock object is an object created to stand in for an object that your code will be collaborating with. Your code can call methods on the mock object, which will deliver results as set up by your tests."

Source: JUnit in Action, Vincent Massol

When are mocks appropriate?

- Real object has non-deterministic behavior
- Real object is difficult to set up
- Real object has behavior that is difficult to cause
- Real object is slow

Source:

http://c2.com/cgi/wiki?MockObjects

Mock Downsides

- Code complexity of being able to switch between real and mock implementations
- Maintaining the mock

Properties of Good Tests

• Automatic	Tests need to be run checked automatically
• Thorough	Test everything that could possibly break
• Repeatable	Tests should produce the same results each time they are run
• Independent	A test should exercise only one thing at a time
• Professional	Tests are code too! Write them professionally.

Source: Pragmatic Unit Testing in C# with NUnit

Recommended Test Organization

- Tests in a separate project/assembly
- Naming conventions

  If you are testing the class:
  Foo.Bar.Baz

  Then you should name your tests
  Foo.Bar.Tests.BazTests
  or
  Foo.Bar.Tests.BazFixture

- Note the namespace and test class name!

Naming Your Tests

- The test’s name should describe the desired outcome and not just be the name of the method under test

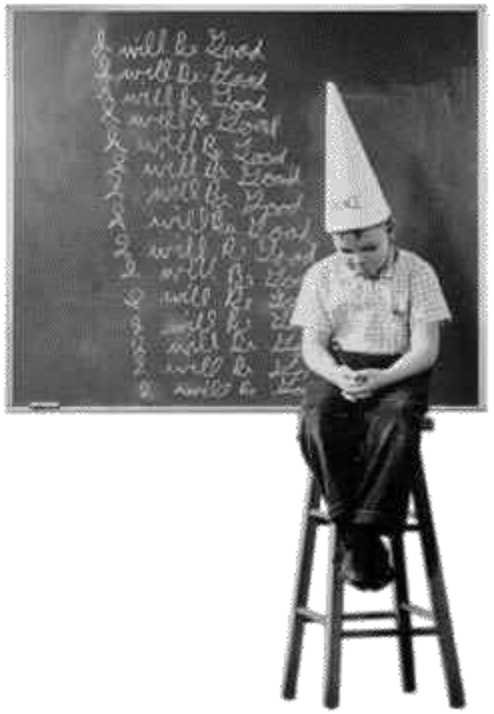
  Good name: PopReturnsLastPushedItem

  Bad name: PopTest

When to Run Tests

- When you write a new method…
  … compile and run local unit tests
- When you fix a bug…
  … run the test that illustrates that bug.
- Any successful compile…
  … run local unit tests.
- Before you check in…
  … run all tests.
- Continuously…
  ... check out and build the project from scratch including all unit tests.

Do not check in code that…



- Is incomplete (e.g. missing dependencies)
- Doesn’t compile
- Compiles but breaks other code
- Doesn’t have unit tests
- Has failing unit tests
- Passes its tests but causes other tests to fail.
- Has tests with the [Ignore] attribute

Additional Resources

Web Sites

- http://www.testdriven.com
- http://www.xprogramming.com
- http://workspaces.gotdotnet.com/tdd

Books

- The Pragmatic Programmer
  Andy Hunt and Dave Thomas
- Test-Driven Development in Microsoft .NET
  Jim Newkirk and Alexei Vorontsov
- Test-Driven Development, by Example
  Kent Beck
- Pragmatic Unit Testing in C# with NUnit
  Andy Hunt and Dave Thomas
- Working Effectively With Legacy Code
  Michael Feathers
- Refactoring
  Martin Fowler
- Lessons Learned in Software Testing
  Cem Kaner, James Bach, and Brett Pettichord