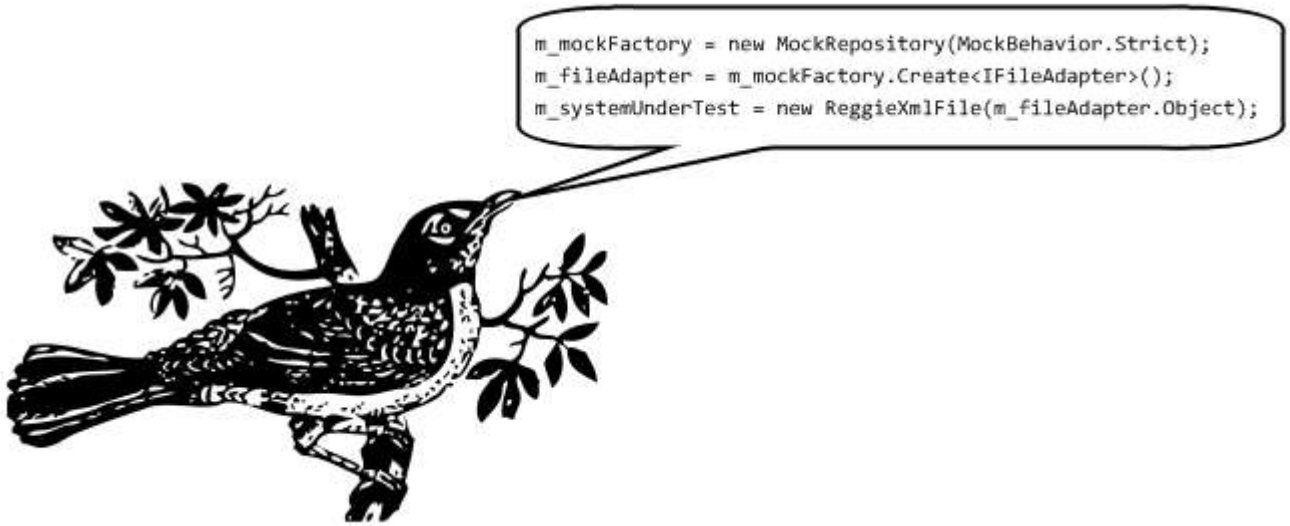
PRINCIPLES AND PATTERNS FOR TEST DRIVEN DEVELOPMENT



Author: Stephen Fuqua
Last Revised: May 2014

PART 1

Testing – Not Just for QA

Benefits: Risk Reduction

- Safely refactor - regression

Benefits: Risk Reduction

- Safely refactor - regression
- Think about "edge cases"

Benefits: Risk Reduction

- Safely refactor - regression
- Think about "edge cases"
- Duh - prove that the software works

Benefits: Better Design

- Safely refactor - evolve

Benefits: Better Design

- Safely refactor - evolve
- Focus on clean, modular code. Key patterns for **easier** testing:
  - Single Responsibility Principle
  - Dependency Injection
  - Adapter

Benefits: Better Design

- Safely refactor - evolve
- Focus on clean, modular code. Key patterns for **easier** testing:
  - Single Responsibility Principle
  - Dependency Injection
  - Adapter
- Concentrate on simple inputs and outputs

Benefits: Better Design

- Safely refactor - evolve
- Focus on clean, modular code. Key patterns for **easier** testing:
  - Single Responsibility Principle
  - Dependency Injection
  - Adapter
- Concentrate on simple inputs and outputs
- Express requirements via tests

Four Types

Scope	**Unit**	**Functional**	Acceptance	Performance
Method	x	x		
Class	x	x		
Includes I/O		x	x	x
Entire Application			x	x
Entire System				x

Where We Are Going

- Thinking About Testing
- Test Driven Development (TDD)
- Legacy TDD
- Obstacles

https://www.flickr.com/photos/wsdot/

PART 2

Thinking About Testing

Approaching Testing

- Write expressively, with intent-revealing names

Approaching Testing

- Write expressively, with intent-revealing names
- Express a business need

Approaching Testing

- Write expressively, with intent-revealing names
- Express a business need
- Concentrate on *inputs* and *outputs* for a *system under test*

Approaching Testing

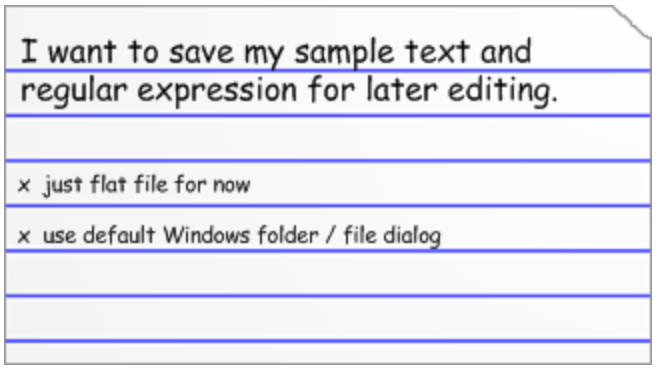
- Write expressively, with intent-revealing names
- Express a business need
- Concentrate on *inputs* and *outputs* for a *system under test*
  - ```
    // Prepare Input
    // Call the system under test
    // Evaluate outputs
    ```
    ```

Approaching Testing

- Write expressively, with intent-revealing names
- Express a business need
- Concentrate on *inputs* and *outputs* for a *system under test*
  - ```
    // Prepare Input
    // Call the system under test
    // Evaluate outputs
    ```
  - ```
    Given
    When
    Then
    ```

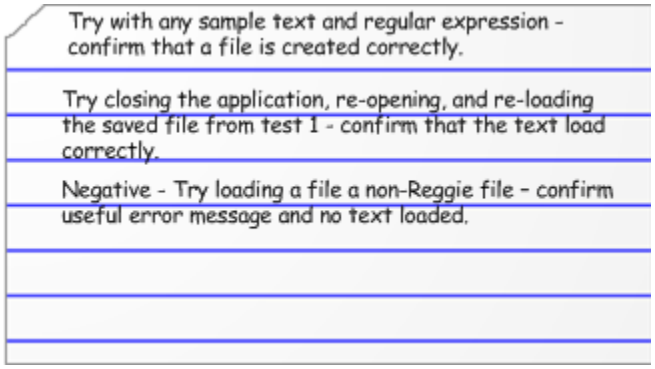
User Stories

- Write (or get) user stories.
- More useful for *Behavior Driven Development*, but still helpful in thinking about *unit* and *functional* tests



User Stories

- Write (or get) user stories.
- More useful for *Behavior Driven Development*, but still helpful in thinking about *unit* and *functional* tests



Negative Testing

- Unexpected input
  - Null values
  - Overflows
- Expected messages
- Exception handling
- Check the logs

Isolation

- Unit: isolated from I/O, web services, etc.

Isolation

- Unit: isolated from I/O, web services, etc.
- Functional: connect to just one outside source

Isolation

- Unit: isolated from I/O, web services, etc.
- Functional: connect to just one outside source
- Avoid interactions from other systems and test

Isolation

- Unit: isolated from I/O, web services, etc.
- Functional: connect to just one outside source
- Avoid interactions from other systems and test
- Setup a self-contained system

Three Essential OO Patterns

- Can’t effectively isolate a method or class without…

Three Essential OO Patterns

- Can't effectively isolate a method or class without…
- **Single Responsibility Principle (SRP)**

Three Essential OO Patterns

- Can't effectively isolate a method or class without…
- Single Responsibility Principle (SRP)
- **Dependency Injection (DI)**
  - Constructor, Property, Method, even Static Delegate Injection

Three Essential OO Patterns

- Can't effectively isolate a method or class without…
- Single Responsibility Principle (SRP)
- Dependency Injection (DI)
  - Constructor, Property, Method, even Static Delegate Injection
- **Adapter**

PART 3

Test Driven Development

Essential Formula

1. **Write a test that fails**

Red

Essential Formula

1. Write a test that fails

Red

2. **Write the code that makes it pass**

Green

Essential Formula

1. Write a test that fails

Red

2. Write the code that makes it pass

Green

3. **Clean up the code**

Refactor

MSTest

```
using System;
using
Microsoft.VisualStudio.TestTools.UnitTesting;

namespace Reggie.UI.Tests
{
    [TestClass]
    public class UnitTest1
    {
        [ClassInitialize]
        public static void
ClassInitializer(TestContext context)
        {
            // Note that it is static.
            // Runs once per class.
        }

        [TestInitialize]
        public void TestInitializer()
        {
            // Runs once before each test.
            // Runs after constructor.
        }

        [TestMethod]
        public void TestMethod1()
        {
        }

        [TestCleanup]
        public void TestCleanuper()
        {
            // Runs once after each test.
        }

        [ClassCleanup]
        public static void ClassCleanuper()
        {
            // Again static.
            // Runs after all tests complete.
        }
    }
}
```

Assertions

- Verify the results after running the system:

```
Assert.<something>(expected, actual, message)
```

- Some frameworks reverse (actual, expected)
- Message – clear enough to know which failed
- Common:
  - Assert.AreEqual
  - Assert.IsNull
  - Assert.IsNotNull
  - Assert.AreSame
  - Assert.IsTrue
  - Assert.IsFalse

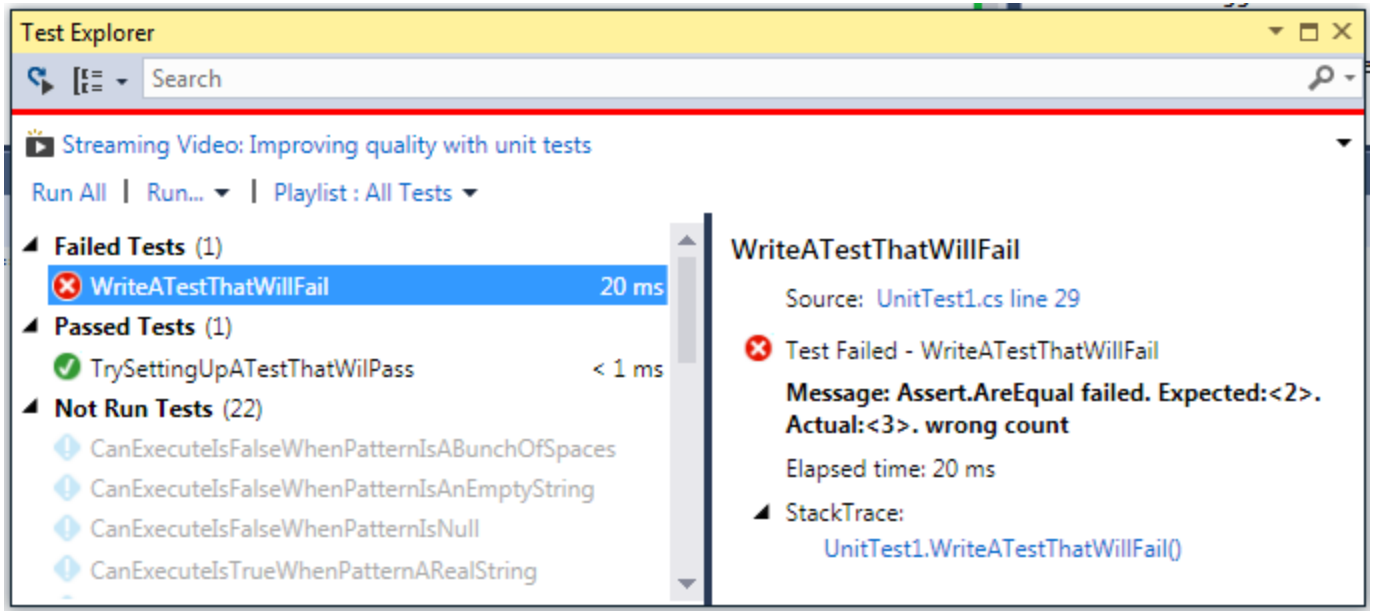
More Verification

- Many frameworks have an attribute like [ExpectedException(typeof(SomeException))]

- Alternately, catch exceptions and inspect details.
- Caught the wrong exception?

```
Assert.Fail(“oops looks like I caught an
”  + typeof(actual).ToString());
```

Test Runner


Test Explorer
Search
Streaming Video: Improving quality with unit tests
Run All | Run... | Playlist : All Tests
Failed Tests (1)
WriteATestThatWillFail 20 ms
Passed Tests (1)
TrySettingUpATestThatWilPass < 1 ms
Not Run Tests (22)
CanExecuteIsFalseWhenPatternIsABunchOfSpaces
CanExecuteIsFalseWhenPatternIsAnEmptyString
CanExecuteIsFalseWhenPatternIsNull
CanExecuteIsTrueWhenPatternARealString
WriteATestThatWillFail
Source: UnitTest1.cs line 29
Test Failed - WriteATestThatWillFail
Message: Assert.AreEqual failed. Expected:<2>. Actual:<3>. wrong count
Elapsed time: 20 ms
StackTrace:
UnitTest1.WriteATestThatWillFail()

Isolation Patterns

- **Fake** – light-weight replacement for expected input or dependency

Isolation Patterns

- **Fake** – light-weight replacement for expected input or dependency
- **Stub** – (partially complete) implementation of an *input* type

Isolation Patterns

- **Fake** – light-weight replacement for expected input or dependency
- **Stub** – (partially complete) implementation of an *input* type
- **Mock** – replacement for input *dependency*, coded to expect specific behavior (*output*)

Isolation Patterns

- **Fake** – light-weight replacement for expected input or dependency
- **Stub** – (partially complete) implementation of an *input* type
- **Mock** – replacement for input *dependency*, coded to expect specific behavior (*output*)
- **Test-specific subclass** – used to *break encapsulation* or provide a *fake*

Mocking

- Could be handwritten
- Typically use a framework: Moq, jMock, Sinon
- Specify only *expected behavior*: in Moq, `MockBehavior.Strict`
- Criticism: over-specifying

Mock Example

```
var mocks = new MockRepository(MockBehavior.Strict);
var filesys = mocks.Create<IFileAdapter>();
var system = new ReggieXmlFile(filesys.Object);

// Prepare input
var input = new ReggieSession()
{
    RegularExpressionPattern = "323",
    SampleText = "46346kljlk"
};
string fileToOpen = "c:\\this\\file.reggie";

// Setup expectations
var extension = ReggieXmlFile.ReggieExtension;
var filter = ReggieXmlFile.ReggieFilter;

filesys.Setup(x => x.OpenFileSaveDialogBox(
      It.Is<string>(y => y == extension),
      It.Is<string>(y => y == filter)))
    .Returns(fileToOpen);
```

```
filesys.Setup(x =>
x.SerializeXmlFile<ReggieSession>(
          It.IsAny<ReggieSession[]>(),
          It.IsAny<string>()))
    .Callback(
        (ReggieSession[] iSession, string iFile) =>
    {
        Assert.AreSame(input,
iSession.FirstOrDefault(), "session");

        Assert.AreEqual(fileToOpen, iFile, "file
name");
    });

// Call the system under test
var actual = system.Save(input);

// Evaluate output
Assert.IsTrue(actual, "wrong response");
mocks.VerifyAll();
```

Mock Example

```
var mocks = new MockRepository(MockBehavior.Strict);
var filesys = mocks.Create<IFileAdapter>();
var system = new ReggieXmlFile(filesys.Object);

// Prepare input
var input = new ReggieSession()
{
    RegularExpressionPattern = "323",
    SampleText = "46346kljlk"
};
string fileToOpen = "c:\\this\\file.reggie";

// Setup expectations
var extension = ReggieXmlFile.ReggieExtension;
var filter = ReggieXmlFile.ReggieFilter;

filesys.Setup(x => x.OpenFileSaveDialogBox(
      It.Is<string>(y => y == extension),
      It.Is<string>(y => y == filter)))
    .Returns(fileToOpen);
```

```
filesys.Setup(x =>
x.SerializeXmlFile<ReggieSession>(
          It.IsAny<ReggieSession[]>(),
          It.IsAny<string>()))
    .Callback(
        (ReggieSession[] iSession, string iFile) =>
    {
        Assert.AreSame(input,
iSession.FirstOrDefault(), "session");

        Assert.AreEqual(fileToOpen, iFile, "file
name");
    });

// Call the system under test
var actual = system.Save(input);

// Evaluate output
Assert.IsTrue(actual, "wrong response");
mocks.VerifyAll();
```

Intent-Revealing Names

```
MockRepository mocks = new
MockRepository(MockBehavior.Strict);
Mock<IFileAdapter> filesys;

[TestInitialize]
public void TestInitializer()
{
    filesys = mocks.Create<IFileAdapter>();
}

[TestMethod]
public void SaveSessionToXmlFile2()
{

    var input = givenASessionObjectStoring(
           pattern: "323",
           text: "46346kljlk");

    string outputFile =
"c:\\this\\file.reggie";
```

```
expectToOpenTheSaveFileDialogBox(outputFile);

expectToSerializeXmlRepresentingThisSession(inp
ut, outputFile);

    var system = givenTheSystemUnderTest();
    var actual = system.Save(input);

    thenTheResponseShouldBe(actual, true);
}

[TestCleanup]
public void TestCleanup()
{
    mocks.VerifyAll();
}
```

Functional Integration Isolation

- Essential: start with a clean slate
- Use a sandbox database on localhost
- Delete and re-create sample files / records
- Launch a service in a separate thread

Code Coverage

Code Coverage

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
sfuqua_QANTAQA 2014-05-21 2...	211	23.09 %	703	76.91 %
reggie.bll.dll	0	0.00 %	18	100.00 %
reggie.bll.tests.dll	1	0.26 %	386	99.74 %
reggie.ui.dll	86	60.56 %	56	39.44 %
{ } Reggie.UI.Properties	5	100.00 %	0	0.00 %
{ } Reggie.UI.Utility	13	100.00 %	0	0.00 %
{ } Reggie.UI.ViewModels	62	52.54 %	56	47.46 %
AboutViewModel	21	100.00 %	0	0.00 %
ReggieBasicViewModel	41	42.27 %	56	57.73 %
{ } Reggie.UI.Views	6	100.00 %	0	0.00 %
reggie.ui.tests.dll	2	0.82 %	243	99.18 %
safnet.systemadapters.dll	122	100.00 %	0	0.00 %

Common Test Smells

- Assertion Roulette
- Interacting Tests
- Conditional Test Logic
- Test Code duplication
- Obscure Test

from xUnit Test Patterns

PART 4

Legacy Testing

Modified Formula

1. **Write a test that passes**

Green

2. Write a test that fails

Red

3. Write the code that makes it pass

Green

4. Clean up the code

Refactor

Refactoring

- Often too hard to test - insufficiently isolated
- Slowly refactor, one step at a time

Refactoring

- Often too hard to test - insufficiently isolated
- Slowly refactor, one step at a time
  - Introduce an interface for constructor injection

Refactoring

- Often too hard to test - insufficiently isolated
- Slowly refactor, one step at a time
  - Introduce an interface for constructor injection
  - Lazy-load for property injection

Refactoring

- Often too hard to test - insufficiently isolated
- Slowly refactor, one step at a time
  - Introduce an interface for constructor injection
  - Lazy-load for property injection
  - Split a method or class into multiple

Refactoring

- Often too hard to test - insufficiently isolated
- Slowly refactor, one step at a time
  - Introduce an interface for constructor injection
  - Lazy-load for property injection
  - Split a method or class into multiple
  - Rethink class variables – pass as arguments instead?

Refactoring

- Often too hard to test - insufficiently isolated
- Slowly refactor, one step at a time
  - Introduce an interface for constructor injection
  - Lazy-load for property injection
  - Split a method or class into multiple
  - Rethink class variables – pass as arguments instead?
  - Test-specific sub-class to set protected variables

Refactoring

- Often too hard to test - insufficiently isolated
- Slowly refactor, one step at a time
  - Introduce an interface for constructor injection
  - Lazy-load for property injection
  - Split a method or class into multiple
  - Rethink class variables – pass as arguments instead?
  - Test-specific sub-class to set protected variables
- Or: brand new code, called from old methods

Fakes and Shims

- Dangerous! Method of last resort!
- Hard-codes dependencies: external resources, MSDN Premium/Ultimate

```
using (ShimsContext.Create())
{
      Fakes.ShimReggieXmlFile.AllInstances.Retrieve
        = (ReggieXmlFile inputFile) =>
            {
                return new Reggie.BLL.Entities.ReggieSession();
            };

    // Now I can test SomeOtherClass that calls the Retrieve method
}
```

Suggestions

- Legacy code deserves tests
- Analyze code coverage for each release, ensuring it goes up
- No emergency maintenance without Green-Red-Green-(Refactor) approach

PART 5

Obstacles

Learning Curve

<discussion>

Learning Curve – Additional Tips

- Resources at end of the presentation
- Study tests in open source projects, e.g. reggie.codeplex.com
- Pair programming

Sufficient Time

<discussion>

Sufficient Time – Additional Tips

- Management must commit
- Double your estimates – then retrospectively check and see if that was “good enough”

Legacy Code

<discussion>

Legacy Code – Additional Tips

- What's the risk tolerance? If high enough, might not be worth it
- Might have better success with BDD than TDD, since BDD typically tests the entire application
- Targeted use of TDD – special cases, enhancements, bug fixes

PART 6

Resources

Books

- *Clean Code*, Robert C. Martin
- *xUnit Test Patterns*, Gerard Meszaros
- *Growing Object-Oriented Software, Guided by Tests*, Steve Freeman and Nat Pryce
- *Agile Testing, A Practical Guide for Testers and Teams*, Lisa Crispin and Janet Gregory
- Can't vouch for personally, but looks promising: *Working with Legacy Code*, by Michael C. Feathers

On The Web

- xUnit Test Patterns (light version of the book)
- Red-Green-Refactor (the original?)
- Martin Fowler on **Mocks Aren't Stubs**
- TDD when up to your neck in Legacy Code
- That Pesky MSTest Execution Ordering…

Author's Blog Posts

- Making Mockery of Extension Methods
- TACKLE: Be Test-Driven
- Dependency Injection with Entity Framework
- Review: Growing Object-Oriented Software, Guided By Tests
- Breaking Down a Unit Test from "Reggie" That Uses MoQ
- Moles: No Longer Fit for Unit Tests
- Breaking My Moles Habit, With MoQ
- Unit vs. Integration Tests When Querying Nullable Columns
- TDD - Scenario for Red, Green, Refactor
- Sub classing for automated testing
- Unit Testing - Code Coverage and Separation of Layers