Unit Testing in Windows

Mark Schofield 2/16/2011

Preamble

- “Unit Testing in Windows”
  - Training provided to Windows Engineers after Windows 7 had shipped
- A few customizations…
  - ‘Internal’ items removed (sorry!)
  - Demo’s still exist
- Questions are OK
  - Let’s chat about stuff; I’ll keep us on track

Who Am I?

- Mark Schofield
  - Lead Software Development Engineer
  - 12+ years at Microsoft
    - 8 years as a “Software Design Engineer in Test”
    - 4+ years owning ‘Test Authoring’ in Windows
- Member of the ‘Engineer Desktop’ team
- Part of the ‘Engineering System’

Agenda

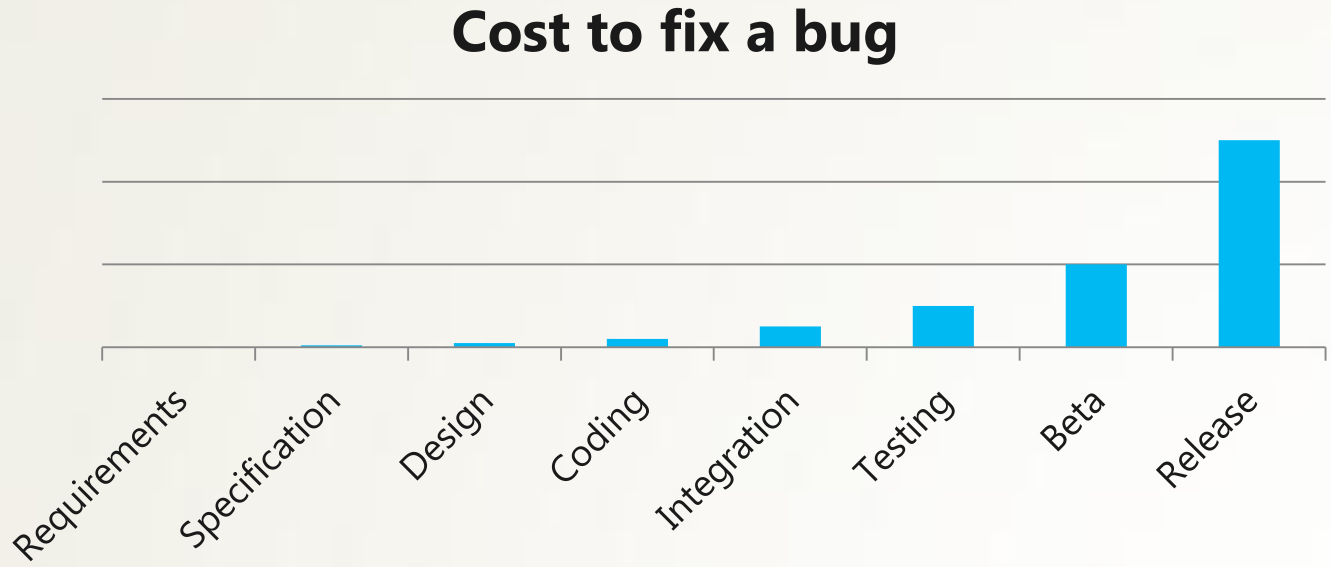
- Setting the Stage
  - Challenges/Opportunities
- Introducing Unit Testing
  - What is a Unit Test
  - Benefits
- Write some Unit Tests
  - Prep
  - TAEF – The "Test Authoring and Execution Framework"
  - Creating your Unit Test binary
- Unit Testing Topics
  - Mitigating Dependencies

Setting the Stage

- Scale
  - ‘Windows’ is big
    - 10’s of thousands of Engineers
    - 100’s of millions of lines of code
    - Source control, branching and versioning means there’s many ‘views’ of the 100’s of millions of lines of code
- Diversity
  - Multiple Languages
    - C, C++, C++/CLI, C#, Assembly Language
    - JScript, Perl, PowerShell

Challenges/Opportunities

- Finding bugs sooner saves money/time



Introduction to Unit Testing

- *"Unit Testing is a relatively inexpensive, easy way to produce better code faster."*
  - Pragmatic Unit Testing, Andy Hunt and Dave Thomas
- Industry practice
  - There's a lot of precedent out there
- Developer Activity
  - Unit Tests shouldn't be 'handed-off' to the Test team

What is a Unit Test?

"A unit test is a piece of a code (usually a method) that invokes another piece of code and checks the correctness of some assumptions afterward. If the assumptions turn out to be wrong, the unit test has failed. A "unit" is a method or function."
- Roy Osherove

Osherove, R. (n.d.). *The Art of Unit Testing: with Examples in .NET.*

So, what *is* a Unit Test?

- Usually a Unit Test exercises some particular method or class in a particular context;
  - Adding a large value to a sorted list; make sure that it's added to the end
  - When manipulating state under certain context, that the correct manipulation happens
- A good starting Unit Test would be to construct a given class, and verify its initial state
  - That's the level that we're working at

Getting everyone on the same page

- If a test requires…
  - …any more than guest privileges…
  - …read/write access to the host operating system's files…
  - …the use of an "install" or "update"…
  - …a "test" operating system to be installed…
  - …crossing process boundaries (including driving UI)…

…it's *not* a Unit Test.

- A Unit Test should run in milliseconds, not seconds.

Why be so strict?

- This definition of Unit Tests is what makes them so valuable
  - Fast, portable, reliable because they're tightly scoped and have no dependencies
  - High 'bang-for-buck' – Developers are working at a level that can leverage their domain expertise
  - Forces good separation, cohesion of code

Benefits of Unit Testing

- "Unit Testing will make your life easier. It will make your designs better and drastically reduce the amount of time you spend debugging."
  – Pragmatic Unit Testing
- You will know sooner and with greater confidence that your code is doing what you intended
- If (or when?) requirements change, you can be more agile in responding to them

Unit Testing isn’t (initially) easy

- Unit Testing may require refactoring of code
  - The code will be better encapsulated and cohesive as a result
  - Writing Unit Tests will encourage Developers to write better code
- Unit Testing is as much about the journey as it is the destination.
- Assigning a single Developer to write a whole team's unit tests is *not* the right approach
- Unit Testing will take 30% of your development time.


by auspices

Let’s get started!


by Nathaniel Robertson

Preparation is important

Cleaning your code

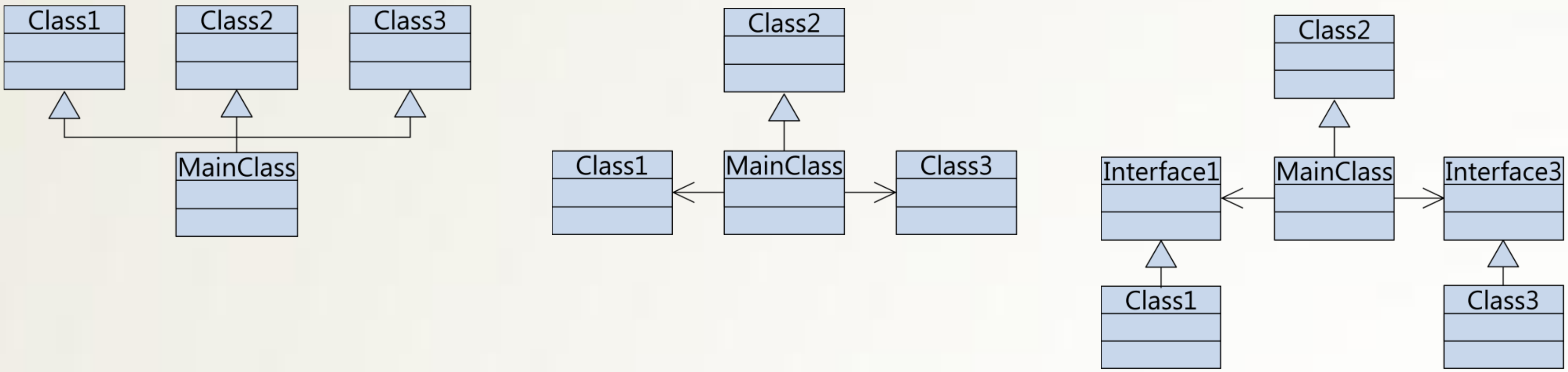
- Declarations go into header files, implementation goes into C/CPP files
  - If you can't #include it, you can't Unit Test it.
  - Increases reusability, too.
- Make header files self-sufficient
  - You'll be compiling it from your product code, *and* your Unit Test code.
- Minimize compile-time dependencies
  - Only #include what you need in the header
  - Forward declarations are OK

An example CPP file

Refactoring your code

- Many of the Design Pattern 'best practices' make code more Unit Testable
  - Prefer minimal classes to monolithic classes
  - Prefer composition to inheritance
  - Avoid inheriting from classes that were not designed to be base classes
  - Prefer providing abstract interfaces
  - Don't give away your internals
- Unit Testing is 'encouraging' better design.
- Herb Sutter's "C++ Coding Standards" is a great reference here.

An example of refactoring



We'll need some tools…

Introduction to TAEF

- Test framework used by Windows Developers and Testers - and other teams across Microsoft
  - Will be shipping in an upcoming Windows Driver Kit
- Foundation for the automation stack; Unit → UI/Scenario
  - Focusing on Developer *and* Tester scenarios
- Evolution of existing tools along with industry practices
  - CppUnit, nUnit, JUnit, xUnit, etc…
- Provides a platform to support different testing methodologies; static, data-driven, etc.

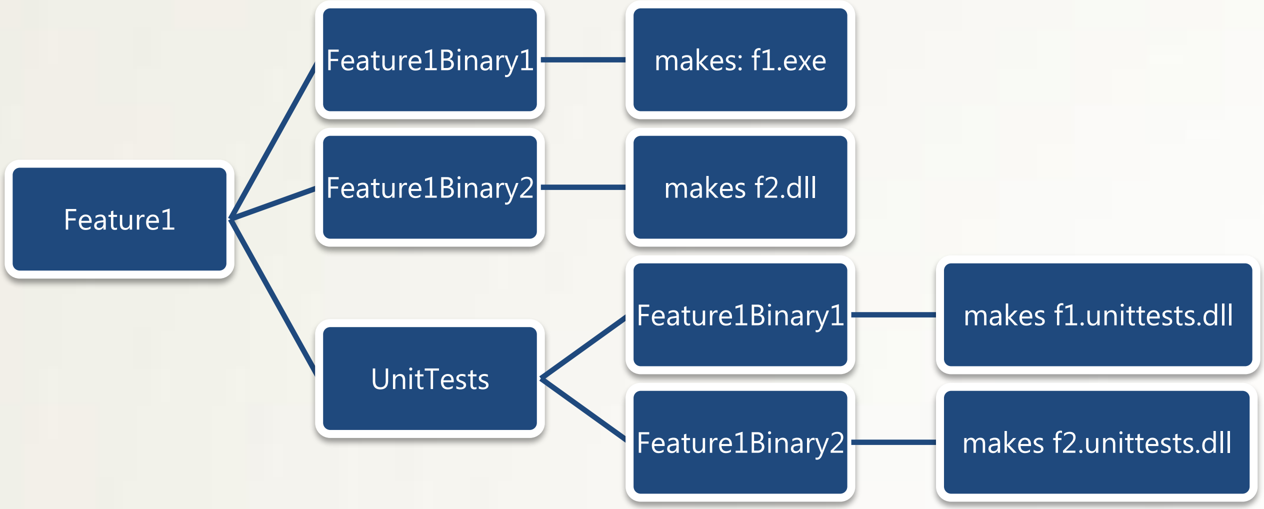
TAEF Features

- No managed or native affinity
  - Teams can use most productive authoring language
    - C/C++, C#, JScript, VBScript
  - Minimal dependencies and pay-for-play features
- 'Out-of-process' execution by default
  - Each 'Test DLL' gets it's own 'sandbox' process.
  - Also supports 'cross-machine' execution.
- Metadata support for selection and runtime environment configuration
- Integration with internal tools

Demonstration

Creating your Unit Test binary

- Source code location
  - In the same project as the product code
  - Under a “UnitTests” folder, following the product code structure:



Creating your Unit Test binary

- DLL Naming
  - "<product binary>.unittests.dll"
    - For example, "notepad.exe" should have Unit Tests in "notepad.unittests.dll"

Authoring a C/C++ Test

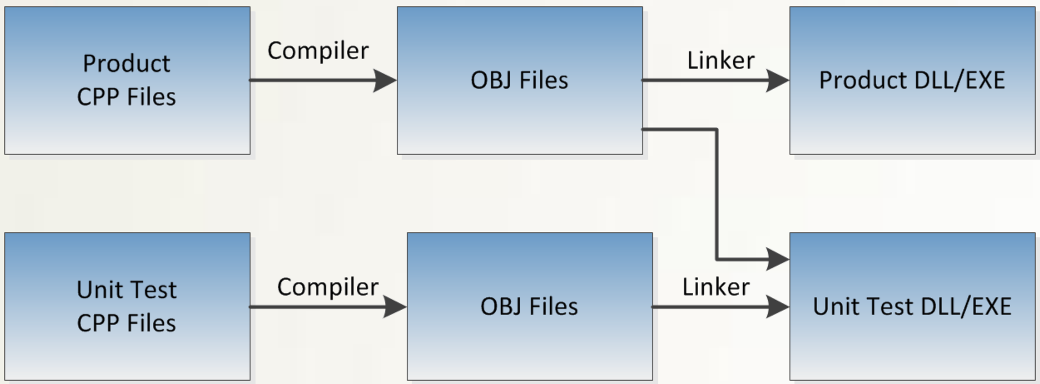
‘MARKING-UP’ THE UNIT TEST

```
#include "WexTestClass.h"
class ManagerTests : public WEX::TestClass<ManagerTests>
{
public:
  TEST_CLASS(ManagerTests)

  TEST_METHOD(ConstructionTests)
  {
    // ...
  }
};
```

Compiling your Unit Test binary

- Native C++ Unit Tests should link directly to the 'obj' files that are produced from the product code.
  - This allows the Unit Tests to interact directly with the product code at the class or function level, without - for example - having to "DLL export" code for it to be visible.



Compiling your Unit Test binary (2)

- DLL exporting the code in order to unit test is not good;
  - It increases the size of the export table of the Product Code binary
  - For classes, exporting the classes restricts the implementation of the class.
  - It increases the surface area of internal APIs
- Don't create a 'lib' of the dll's product code just for Unit Testing
  - It's an extra build step that's unnecessary

Writing Unit Tests

- Start simple
  - The first test that you write should be incredibly simple, to make sure that you can create, compile and run it.
- The general pattern for the Unit Test code:
  - Set-up all conditions needed for testing
  - Call the method to be tested
  - Verify that the tested method functioned as expected
  - Cleanup anything it needs to

Writing Unit Tests

- The 'VERIFY' macros helps verify the state that you expect
  - Effortless verification/logging APIs; encourages a consistent logging pattern
  - Logs concise message if verification succeeds; more detailed (type-aware) message if verification fails.
  - Streamlines test code by removing the need to nest verification calls (if compiled with C++ exceptions enabled).
- You'll get concise output on success, detailed output on failure

'Verify' examples

- Write:

```
VERIFY_ARE_EQUAL(myExpectedValue, MyFirstTestFunction());
VERIFY_SUCCEEDED(MySecondTestFunction());
```

- As opposed to:

```
int result = MyFirstTestFunction();
if (result == myExpectedValue)
{
    Log::Comment("MyFirstTestFunction() succeeded");
    HRESULT hr = MySecondTestFunction();
    if (SUCCEEDED(hr))
        Log::Comment("MySecondTestFunction() succeeded");
    else
        Log::Error("MySecondTestFunction() did not return the expected result");
}
else
    Log::Error("MyFirstTestFunction() did not return the expected result");
```

Writing Unit Tests

- Unit Tests should be very linear
  - Little – if any – control flow
  - If there's control flow; should it be a different test?
- Code for the success case
  - Production code needs to accommodate all scenarios, failures, error cases, edge cases, etc, unit test code doesn't
- Unit Tests should be quick to write
  - Test Harness should support this, by having a low 'per test' overhead

Demonstration

Running Unit Tests

- Using TAEF:

```
te UIAnimation.unittests.dll
```

- Select the right tests to get quick verification:

```
te UIAnimation.unittests.dll /select:@Name='ManagerTests::*'
te UIAnimation.unittests.dll /name:ManagerTests::*
```

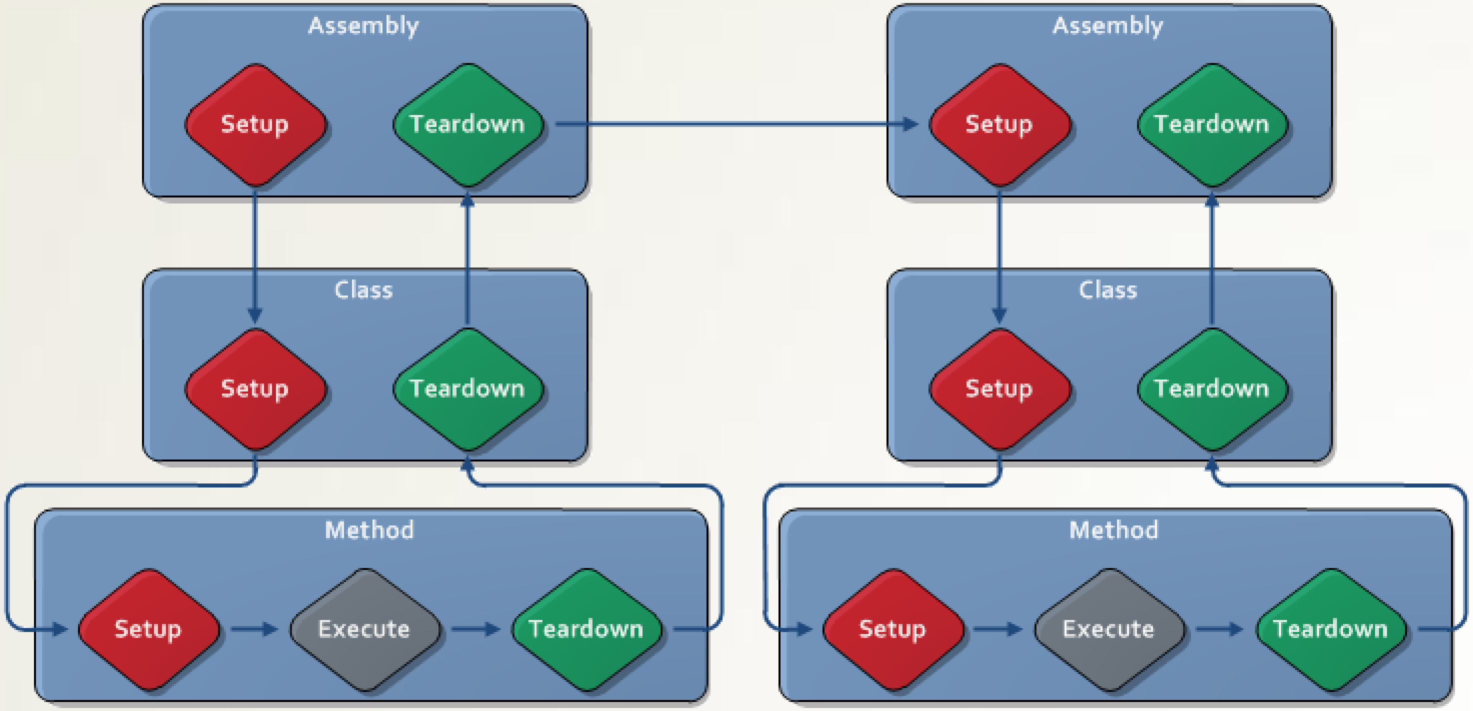
- The selection language allows you to select through metadata, using ‘and’, ‘or’ and ‘not’ semantics.

Demonstration

Setup and Cleanup

- Like most Unit Test harnesses, TAEF supports Setup and Cleanup ‘fixtures’ to allow shared code to ‘bookend’ tests
- You can write fixtures around Tests, a Class or a DLL
- TAEF guarantees that the fixtures are prepared before the test is run
  - All fixtures run on the same thread as the test itself.

Setup and Cleanup



Adding metadata

- Metadata is simple data associated with the test code.
- Metadata can be applied to DLL's, Classes or Tests
- Metadata is 'inherited'
- Metadata is used for:
  - Selection
  - Runtime environment configuration

Adding metadata (2)

- ‘Marking-up’ the Unit Test:

```
#include <WexTestClass.h>
class VariableTests : public WEX::TestClass<VariableTests>
{
public:
    BEGIN_TEST_CLASS(VariableTests)
        TEST_CLASS_PROPERTY(L"Owner", L"MSchofie")
    END_TEST_CLASS()

    TEST_METHOD(ConstructionTests);
    TEST_METHOD(ValueChangeTests);
};
```

Demonstration

Mitigating dependencies

- The most difficult aspect of Unit Testing is ‘mitigating dependencies’
- Unit Tests need to execute the ‘unit’ in isolation
  - Dependent methods or objects should be replaced (somehow) with a ‘test double’.
  - Test Double: A test specific equivalent of product code.
- There’s different ways to solve this
  - Techniques differ based on the language, level, practicality, cost

Mitigating dependencies

THE GAMUT OF TECHNIQUES

- Design-time
  - Use Design Patterns to allow the introduction of a Test Double at Unit Test-time
- Compile-time
  - Compile different implementations into the product code, when compiling the code into your Unit Tests
- Link-time
  - Link to test doubles functions, control behavior at runtime
- Run-time
  - Change/replace the implementation at runtime

Design-time Mitigation

- Use of design patterns decouples implementation through interfaces
  - "Program to an Interface, not an Implementation"
- Interfaces provide a great opportunity for introducing test doubles
- Unit test can declare a function scoped class that implements the specific interface

Design-time Mitigation

```
class ComplexSystem
{
public:
    ComplexSystem(IDependency& dependency, int parameter) :
        m_dependency(dependency)
    {
        m_dependency.Initialize(parameter);
        // ...
    }
private:
    IDependency& m_dependency;
};
```

Compile-time Mitigation

- Often ‘cheaper’ than design-time mitigation
  - Less work
  - More performant than ‘design-time’ mitigations
    - Compile-time polymorphism, not runtime polymorphism
- Uses C++ techniques
  - Not suitable for C
  - May require moving code into headers

Compile-time Mitigation

EXAMPLE – DEPENDENT CLASS

```
class ComplexSystem
{
public:
    ComplexSystem(int parameter) : d(parameter)
    {
        // ...
    }
private:
    DependentClass d;
};
```

Compile-time Mitigation (2)

EXAMPLE – DEPENDENT CLASS

```
template <typename TDependentClass = DependentClass>
class ComplexSystemT
{
public:
    ComplexSystemT(int parameter) : d(parameter)
    {
        // ...
    }
private:
    TDependentClass d;
};
typedef ComplexSystemT<> ComplexSystem;
```

Compile-time Mitigation (3)

EXAMPLE – DEPENDENT CLASS

- Within the test, provide a function scoped double, and provide that to the template class

```
TEST_METHOD(ComplexSystemTest)
{
  class DoubleDependentClass
  {
    // ...
  };
  ComplexSystemT<DoubleDependentClass> system;
}
```

Runtime Mitigation

- An internal library – 'Mock10' – provides support for replacing function and method implementations at runtime.
  - Uses 'Detours' a library that Microsoft Research owns: http://research.microsoft.com/en-us/projects/detours/
  - Provides a high-level, C++ API for replacing functions
  - It's C++0x aware – supporting Lambda's
  - Supports filtering based on calling frame, calling module and parameters

Runtime Mitigation (2)

- Allows users to write code like:

```
auto mock = Mock::Function(::CreateFileW, [] (/* ... */) -> HANDLE
    {
        ::SetLastError(ERROR_PATH_NOT_FOUND);
        return INVALID_HANDLE_VALUE;
    });
```

Demonstration

Summary

- Introduced Unit Testing
- Wrote some Unit Tests
  - Used metadata for selection
  - Used 'fixtures' for code reuse
- Mitigated dependencies
  - Design-time, compile-time, run-time techniques

Questions?

Thank you.