

Unit Testing with .NET



(C) 2002, 20th Century Fox, Minority Report

Michael Stal, Senior Principal Engineer
Siemens Corporate Technology, CT SE 2
E-mail: Michael.Stal@siemens.com



Software &
Engineering
Architecture

Agenda

- Motivation of Unit Testing
- Do it Yourself
- Introducing NUnit by Example
- NUnit in Detail
- Mock Objects
- Test your Design – Design your Test
- Test First Reloaded
- Some Recommendations
- Summary



Software &
Engineering
Architecture



Software &
Engineering
Architecture

Code First

- **A few years ago testing was considered the sole responsibility of an alien race of creatures called testers**
- **Developers often revealed a „that’s not my job“ or „I better use my time for writing code“ habit**
- **Especially, developers were not in charge of most test activities:**
 - *Integration Test*: useful to show and resolve problems if different developer groups develop different parts of a system that must co-operate
 - *System Test*: testing the system as a whole from a more black box perspective
 - *Acceptance Test*: testing if the system under development is what customers expect it to be, basically a black-box test
 - *Stress Test*: push the limits of the system until failures occur
 - *Beta testing*: push the limits of consumers until failures occur ☺

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
3



Software &
Engineering
Architecture

Technologies, Architecture, Processes

- **In the last years, however, speed of development is rapidly increasing (time-to-market)**
- **At the same time technologies keep constantly changing and evolving**
- **Quality is still important, i.e.**
 - Functional properties
 - Non-functional properties (developmental+operational)
- **If everything else is unstable, two aspects are essential:**
 - clean architectural design to provide a sufficient base for quality
 - intensive testing to check and prove quality
- **Processes have to adapt to this development: agile processes (e.g., Scrum, XP) and incremental approaches such as RUP**

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
4

Test Driven Development

- In this context a big-bang approach (big design upfront) does not work anymore
- Instead, software testing must accompany the whole development process and developers must actively participate
- Live the mantra of agile development: design a little, code a little, test a little
- We as developers must provide tests to check that our code does the right things correctly
- Writing software tests becomes part of our jobs
- Unit tests means testing small pieces of functionality
 - Tests focus on small fractions of code (e.g. methods)
 - They exercise code to prove code does what it is supposed to do

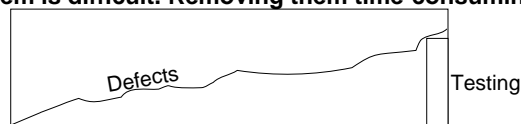


Software &
Engineering
Architecture

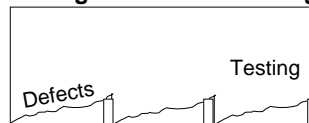
© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
5

Why Unit Testing

- Without unit tests defects show up in the end of development
- Finding them is difficult. Removing them time-consuming and complex



- With unit tests we can find defects much sooner and easier, continuously removing them before adding new features



- We can also check if new features break our code



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
6

Automating Tests with Test-Driven Development

- In each increment only a (small) feature (set) is developed
- Unit tests are included so that we can check whether the actual code meets its requirements. Unit tests are also checked into the project repository
- Unit tests are automatically triggered by the build environment or may also be triggered by a button click in a Unit Test GUI
- If the tests succeed we as developers gain confidence in our work
- Note: Integration tests may also consist of unit tests
- On errors and test failures we refactor our code until all unit tests succeed
- In further increments code and maybe also unit tests are extended or sometimes even modified



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
7

Test First Approach

- You might first write the code and then a test to check what you got
- In a Test First Approach you first write the test and then the code in such a way that the test succeeds:
 1. Define one small property your code should have
 2. Write a unit test that fails
 3. Write only those parts of the code that make the test successful
 4. Go to 1 unless all features available
- The Unit Test becomes the specification of your code and an appropriate means for documenting your work



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
8



Do it Yourself – Example (1)

- Let us implement our own stack of integers – a class that the world has been waiting for

```
public class IntStack {
    int[] _stack;
    int _stackp;
    int _capacity;
    public IntStack(int capacity)
    {
        if ((capacity = _capacity) <= 0)
            throw new ArgumentOutOfRangeException();
        _stack = new int[_capacity];
        _stackp = 0;
    }
    public int Top() { return _stack[_stackp-1]; }
    public void Push(int val) { _stack[_stackp++] = val; }
    public int Pop() { return _stack[--_stackp]; }
    public bool IsEmpty() { return _stackp == 0; }
    public bool IsFull() { return (_stackp > _capacity); }
    public void Clear() { _stackp = 0; }
}
```



Do it Yourself – Example (2)

- Now we can test step by step different aspects of our class
- First make sure creation succeeds:

```
IntStack istack = new IntStack(3);
```

- Oops, we get an `ArgumentOutOfRangeException` due to an error in our code:

```
int _capacity;
public IntStack(int capacity)
{
    if ((capacity = _capacity) <= 0)
        throw new ArgumentOutOfRangeException();
}
```

- We have mixed member variable and argument in the if statement. So let us fix this.
- Now the first test succeeds

Do it Yourself – Example (3)

- Now let us check step by step other parts of our code:

```

IntStack i stack = new IntStack(3);
Debug.Assert(i stack. IsEmpty());
i stack. Push(1);
Debug.Assert(! i stack. IsEmpty());
Debug.Assert(i stack. Top() == 1);
Debug.Assert(i stack. Pop() == 1);
Debug.Assert(i stack. IsEmpty());
i stack. Push(42);
i stack. Clear();
Debug.Assert(i stack. IsEmpty())

```

- Using assertions we check all of our operations: does the return value and behavior match our expectation?



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
11

Do it Yourself – Example (4)

- We might also check exceptions:

```

try {
    new IntStack(-1);
    // failure if we reach this line:
    Debug.Assert(false);
}
catch (ArgumentOutOfRangeException) {}

```

- In other word we expect an `ArgumentOutOfRangeException` here and our code is exactly throwing this exception



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
12

Do it Yourself – Example (5)

- We should also push our class to its limits:

```

i stack = new IntStack(3);
i stack. Push(1);
i stack. Push(2);
i stack. Push(3);
Debug. Assert(i stack. IsFull ());

```

- We expect no failure but IsFull() does not work correctly. Let us refactor our code:

```

public bool IsFull () { return (_stackp > _capacity); }
=>
public bool IsFull () { return (_stackp == _capacity); }

```

- Now this and the other tests succeed



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
13

Do it Yourself – Example (6)

- Finally, we should check what happens if we use Push and Pop too often. We expect InvalidOperationException in this case, e.g.:

```

i stack = new IntStack(3);
i stack. Push(1);
i stack. Push(2);
i stack. Push(3);
try {
    i stack. Push(4);
    Debug. Assert(fal se);
} catch (InvalidOperationException) {}

```

- Oops, we are getting an IndexOutOfRangeException instead!



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
14



Do it Yourself – Example (7)

- We forgot in our implementation to check for index violations.
Lets refactor again:

```
public class IntStack {
    int[] _stack; int _stackp; int _capacity;
    public IntStack(int capacity) {
        if ((_capacity = capacity) <= 0)
            throw new ArgumentOutOfRangeException();
        _stack = new int[_capacity]; _stackp = 0;
    }
    public int Top() {
        if (!IsEmpty()) return _stack[_stackp-1];
        else throw new InvalidOperationException(); }
    public void Push(int val) {
        if (!IsFull()) _stack[_stackp++] = val;
        else throw new InvalidOperationException(); }
    public int Pop() {
        if (!IsEmpty()) return _stack[--_stackp];
        else throw new InvalidOperationException(); }
    public bool IsEmpty() { return _stackp == 0; }
    public bool IsFull() { return (_stackp == _capacity); }
    public void Clear() { _stackp = 0; }
}
```



Lessons Learned

- In this manually crafted unit test code we checked different parts of our code for conformity with our expectations
- We used an executable test program containing Debug.Assert statements for checking the code
- In each individual test we set up the environment necessary for the test, called the class under test, and then checked for failures
- Expected exceptions were handled by try/catch blocks and inserting Debug.Assert(false) after the code raising the exception
- This approach was very helpful but neither efficient nor intuitive convenient, systematic, automated
- We need support: A Unit Test Framework is what we really need

NUnit Framework

- NUnit denotes a .NET Unit Test Framework
- It follows an idiomatic approach instead of just cloning JUnit (first Unit Framework written by Kent Beck and Erich Gamma)
- To use NUnit get the current version from <http://nunit.sourceforge.net>
- Install it if your IDE does not already integrate NUnit (e.g., this is the case in Whidbey/VIS.NET 2005 Beta)
- There is an additional VS.NET add-in available
- NUnit comes with different test-runners, one that offers an interactive GUI and a console-based version
- Include reference to `nunit.framework.dll` to your test code and use namespace `NUnit.Framework`



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
17

NUnit Example (1)

- Here is an example test that checks our `IntStack`. We set up a new library project (DLL) for this purpose:

```
using System;
using NUnit.Framework;
using MyCollections; //namespace of IntStack

namespace TestIntStack
{
    [TestFixture] //this class defines tests
    public class IntStackUnitTestClass
    {
        IntStack iStack;
        int capacity;
```



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
18

NUnit Example (2)

- We can use special methods for setting up and tearing down the test environment.
- TearDown is not shown here
- NUnit will call set-up and tear-down for each test method

```
[SetUp]
public void Init()
{
    capacity = 3;
    iStack = new IntStack(capacity);
}
```



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
19

NUnit Example (3)

- Each test method exercises one specific aspect of the class under test. Test methods are annotated with [Test]
- The first test checks whether creation and initialization are successful

```
[Test]
public void EmptyTest()
{
    Assert.IsTrue(iStack.IsEmpty());
}
```



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
20

NUnit Example (4)

- In another test we check that the implementation will throw an appropriate exception if we try to pop from an empty stack

```
[Test,
ExpectedException(typeof(InvalidOperationException))
public void PopFromEmptyStack() {
    iStack.Pop();
}
```

- Step by step we continue defining our test methods
- Now we should automatically run the tests using the GUI (or the command line console)

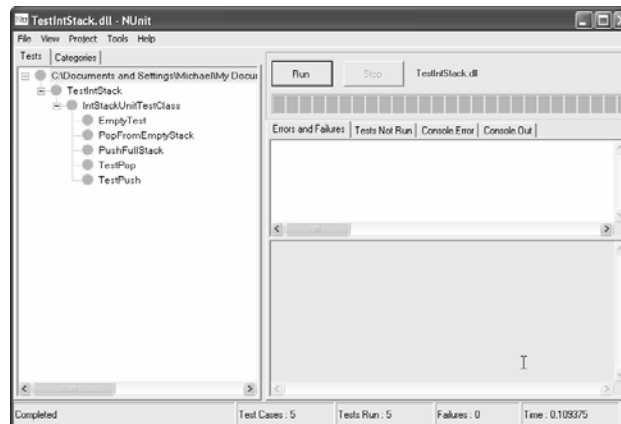


Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
21

NUnit Example (5)

- Now, we are able to use the GUI test runner for checking our class. Green is the color of success



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
22

NUnit Example (6)

- We can run the same test using the console application (nunit-console.exe) or even in VS.NET with the right add-in
- This tool doesn't offer a nice UI but can be easily integrated in your build environment (e.g., CruiseControl.NET or NAnt)

```

C:\WINDOWS\system32\cmd.exe
C:\Documents and Settings\Michael\My Documents\Visual Studio Projects\ClassLibrary8\MyCollections\bin\Debug>c:\program files\nunit 2.2\bin\nunit-console" mycollections.dll
NUnit version 2.2.1
Copyright (C) 2002-2003 James W. Newkirk, Michael C. Two, Alexei A. Vorontsov, Charlie Poole,
Copyright (C) 2000-2003 Philip Craig.
All Rights Reserved.

OS Version: Microsoft Windows NT 5.1.2600.0 .NET Version: 1.1.4322.2032

****
Tests run: 4, Failures: 0, Not run: 0, Time: 0.046875 seconds

C:\Documents and Settings\Michael\My Documents\Visual Studio Projects\ClassLibrary8\MyCollections\bin\Debug>

```



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
23

NUnit

- Using NUnit for unit testing always involves the following activities:
 - Set up test environment (required objects,resources, ...)
 - Call method to be tested
 - Verify result with your expectations
 - Clean up
- The test library developed by yourself and executed by NUnit is a separate DLL that tests the production code
- NUnit parses through you test code and uses reflection to determine what to do



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
24

JUnit Assert Statements

- **Various assert statements are provided by the framework. If assertion fails, the test fails:**

- `AreEqual (expected, actual [, string message])` tests if the expected value and the actual value are equal. Message will be displayed on error
- `IsNull (object [, string message]), IsNull (object [, string message])` check for null-ness
- `AreSame (expected, actual [, string message])` checks if references are equal (does not check equal contract!). Also available: `AreNotSame`
- `IsTrue [bool cond, string message]`, check whether condition is true. Also available: `IsFalse`
- `Fail ()` fails tests immediately and is not often used



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
25

Test Methods

- **Each test method is annotated with the attribute [Test]**
- **Focus on a specific part in each test method**
- **Use asserts to verify**

```
[Test]
public void TestPop() {
    i stack.Push(1);
    i stack.Push(2);
    Assert.AreEqual (2, i stack.Pop());
    Assert.AreEqual (1, i stack.Top());
}
```

- **In contrast to some literature there need not be a 1:1 mapping between test method and method of class under test**



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
26

Dealing with Exceptions

- **With respect to exceptions there are two issues:**

- Exceptions you expect should be specified in an attribute:

```
[Test,
ExpectedException(typeof(InvalidOperationException))]
public void PopFromEmptyStack() {
    i stack.Pop();
}
```

- Exceptions you don't expect will be intercepted by NUnit and reported as an error. I provoked an error by inserting division by zero:

```
TestCase '
TestIntStack.IntStackUnitTestClass.TestPush'
failed:      System.DivideByZeroException :
Attempted to divide   by zero ...
```



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
27

NUnit Test Categories (1)

- **In NUnit you may specify test categories by annotating the Test Attribute with a category parameter:**

```
...
[Test, Category("PushTests")]
public void TestPush() {
    i stack.Push(1);
    Assert.AreEqual(1, i stack.Top());
}
[Test, Category("PopTests"), Explicit]
public void TestPop() {
    i stack.Push(1);
    i stack.Push(2);
    Assert.AreEqual(2, i stack.Pop());
    Assert.AreEqual(1, i stack.Top());
}
...
```

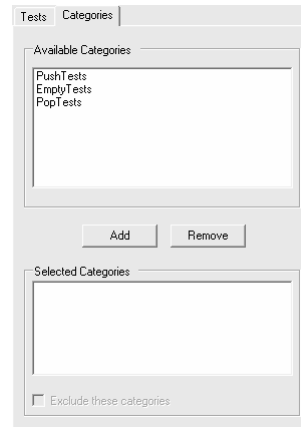


Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
28

NUnit Test Categories (2)

- In the test runner you can include or exclude categories
- Might also be used to differentiate between tests that are slow and should only run once per day/week and fast tests
- If no category is selected all tests will be executed except of those annotated with an explicit attribute

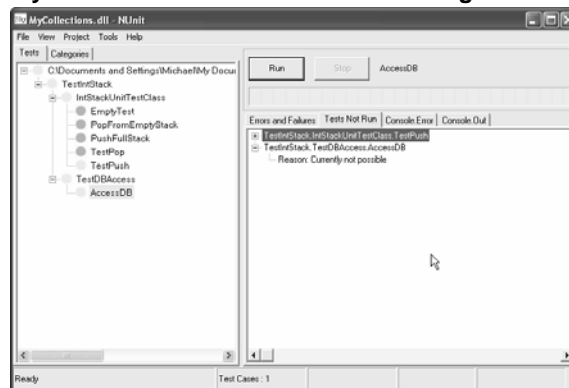


Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
29

Ignoring Tests

- Using the attribute `Ignore` you might instruct the test runner not to run a particular test. This is interesting to specify tests which can currently not be executed due to code missing or other reasons



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
30

Test SetUp and TearDown (1)

- You might specify one SetUp and one TearDown method per TestFixture
- SetUp is executed before each individual test method. Then the test method is executed. Then TearDown gets called.

```
[TestFixture]
public class TestDBAccess {
    int session;
    [SetUp]
    public void Init() {
        session = DBConnect("...");
    }
    [TearDown]
    public void Fini () {
        DBClose(session);
    }
    // and now the tests: not shown!
```



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
31

Test SetUp and TearDown (2)

- SetUp is executed before each individual test. Then the test is executed. TearDown gets called after the test
 - SetUp
 - Test1
 - TearDown
 - SetUp
 - Test2
 - TearDown
 - ...
- For Per-Class SetUp and TearDown use the attributes [TestFixtureSetUp] and [TestFixtureTearDown]



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
32

Custom Asserts

- You can write your custom classes to add new assertion methods.
- For instance,

```
using NUnit.Framework;

public class GUIAssert {
    public static void AreSameColor
    (byte red, byte green, byte blue, Color c)
    {
        Assert.AreEqual(c.r_value, red);
        Assert.AreEqual(c.g_value, green);
        Assert.AreEqual(c.b_value, blue);
    }
    ...
}
```

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
33



Software &
Engineering
Architecture

Custom Asserts – Inheritance Approach

- You might also derive from `NUnit.Framework.Assert` and include your custom assertions in the derived class:

```
public class MyAssert : Assert
{
    public static void IsBefore(DateTime d1,
                               DateTime d2)
    {
        Assert.IsTrue(d1.CompareTo(d2) < 0);
    }
}

// sample usage:
DateTime dt = ... // get time when file was
modified
MyAssert.IsBefore(dt, DateTime.now);
```

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
34



Software &
Engineering
Architecture

Using Files for Test Data

- Sometimes you want to test a complete series of input data
- The input data should be manually changeable
- For such scenarios a solution is to read test data from a file, e.g.,
 - XML documents
 - Excel Files
 - Databases
 - Self-defined files
- The data is read from the file and then used to execute the test
- We will see an example in the next slides



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
35

File Input Example (1)

- The generic class `Collector<T>` allows to apply an operation on all members of a collection, collecting these results, starting with an initial value, e.g.:
 - Adding all elements of an int array, starting with 0
 - Concatenating all strings of a string collection starting with „“

```
namespace CollectOperation {
    public class Collector<T> {
        public delegate T CollectDelegate(T temp, T parm);
        public static object DoCollect
            (ICollection<T> coll, CollectDelegate cdel, T ini tVal)
        {
            T res = ini tVal;
            foreach (T elem in coll) res = cdel(res, elem);
            return res;
        }
    }
}
```



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
36



File Input Example (2)

- **The test is setup with all required delegates and helper methods:**

```
using System.IO;
using NUnit.Framework;
using System.Collections;
using CollectOperation;

namespace UnitTestCollectOperation {
    [TestFixture]
    public class TestUnit {
        static Collector<int>.CollectDelegate addDel;

        // helper method
        public int Add(int res, int parm) {return res + parm;}

        [SetUp]
        public void Init() {
            addDel = new Collector<int>.CollectDelegate(Add);
        }
    }
}
```



File Input Example (3)

- **The test operation itself:**

```
[Test]
public void CheckAddOperations() {
    String line;
    StreamReader reader = new StreamReader(@"c:\testdata.txt");
    while ((line = reader.ReadLine()) != null) {
        if (line.StartsWith("//") || (line==String.Empty)) continue;
        line = line.Replace(" ", "");
        String[] parsed = line.Split('-');
        String res = parsed[0]; // here we read the operation
        switch (res) {
            case "ADD":
                int expected = Int32.Parse(parsed[1]);
                int ini tValue = Int32.Parse(parsed[2]);
                ArrayList al = new ArrayList();
                for (int i=3; i<parsed.Length; i++) al.Add(Int32.Parse(parsed[i]));
                int[] valuelist = (int[]) al.ToArray(typeof(int));
                Assert.AreEqual (expected, Collector<int>.DoCollect(valuelist, addDel,
                ini tValue));
                break;
            default: // omitted for brevity
                break;
        }
    }
}
```

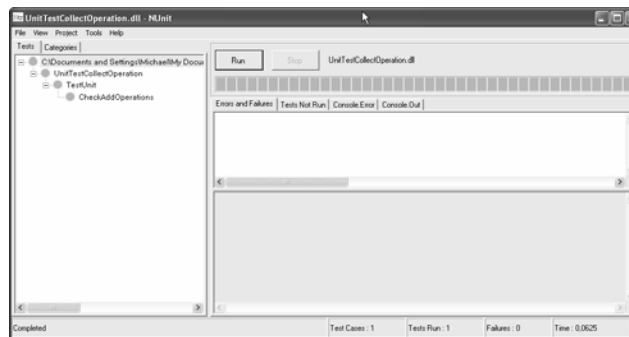


Software &
Engineering
Architecture

File Input Example (4)

- The test data file contains:

```
// Operation - Expected Result - Init Value - List of Operands
ADD - 12 - 0 - 3 - 2 - 4 - 3
ADD - 15 - 0 - 1 - 2 - 3 - 4 - 5
ADD - 16 - 1 - 1 - 2 - 3 - 4 - 5
```



© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
39



Software &
Engineering
Architecture

Testing ASP.NET Applications (1)

- For ASP.NET unit testing developers might use NUnitAsp. Using NUnit would mean to add a lot of interception and facade code
- Version 1.5.1 based on NUnit v2.2
- Don't need to know how ASP.NET engine renders HTML because that's hidden
- ASP.NET pages still run in ASP.NET server engine but
 - a facade container in client code checks the UI interactions and
 - a browser imitation object triggers user interaction
- Get it from <http://nunitasp.sourceforge.net/>
- Tutorial available at <http://www.theserverside.net/articles/showarticle.tss?id=TestingASP>

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
40



Testing ASP.NET Applications (2)

- **Example taken from NUnitAsp Web Page:**

```
[Test]
public void TestExample()
{
    // First, instantiate "Tester" objects:
    LabelTester label = new LabelTester("textLabel",
    CurrentWebForm);
    LinkButtonTester link = new LinkButtonTester("linkButton",
    CurrentWebForm);

    // Second, visit the page being tested:
    Browser.GetPage("http://localhost/example/example.aspx");

    // Third, use tester objects to test the page:
    AssertEquals("Not clicked.", label.Text);
    link.Click();
    AssertEquals("Clicked once.", label.Text);
    link.Click();
    AssertEquals("Clicked twice.", label.Text);
}
```



Unit Test and Environment Issues

- **Unit tests should be independent of each other**
- **They also should not depend on environment issues**
- **Otherwise, it ain't possible to check individual functionality separately**
- **But what if your class under test depends on parts of the environment that**
 - do not yet exist
 - have non-deterministic behavior (e.g., sensors that show temperature)
 - are not easy to trigger (e.g. all kinds of failures)
 - are very slow
 - represent a user interface (web based or windows based)
 - can not be queried easily to see what happened ???

Mock Objects

- To deal with the aforementioned issues we must be able to provide the environment a class under test expects
- If we cannot use the physical objects we must provide mock objects with the same interface
- There are different options:
 - Build the mock objects from scratch
 - Derive Mock subclasses from real object base classes
 - Build an interface which both the mock objects and real objects implement
- One warning, however: If mock objects do not model their real counterparts correctly, test results may fool you



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
43

Do it Yourself – Example (1)

- Suppose we want to build a client application that allows to automatically buy and sell shares when rules apply

```
public class ShareClient {
    // a lot of stuff omitted
    public string Credentials { get { return _credentials; } }
    public double Credits { get { return _credits; } }
    public void AddRule(Option what, string shareName, int howMany, double limit)
    { /* ... */ }
    public void RemoveRule(string shareName) { /* ... */ }
    public ShareClient(ShareNotificationServer srv, string cred, double credit)
    {
        // .. client registers with Server ...
        _srv.RegisterClient(_cred, _credit);
        _srv.OnShareChanged += new ShareNotificationServer.ShareChanged(OnChange);
    }
    private void OnChange(object src, ShareNotificationServer.ShareChange sc)
    {
        // gets notified whenever share changes and applies rules
        // example: _srv.Transaction(_cred, sc.ShareName, Option.BUY, r.howMany);
    }
}
```



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
44

Do it Yourself – Example (2)

- **Unfortunately,**
 - the server isn't yet implemented
 - would be slow and non-deterministic even if it were available
- **Nonetheless, we want to build our client code and test it independently**
- **This is a situation where mock objects are helpful**
- **So let us build a Mock Object with exactly the interface the real server would provide**



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
45

Do it Yourself – Example (3)

- **Here is the public interface of our server:**

```
public enum Option { SELL, BUY };
public class ShareNotificationServer {
    public struct ShareChange { }
    public struct Share { }
    public class ClientAlreadyRegisteredException: ArgumentException { }
    public class ClientUnknownException: ArgumentException { }
    public class ShareUnknownException: ArgumentException { }
    public class TransactionFailure: ArgumentException { }
    public ShareNotificationServer() { }
    public void RegisterClient(string myCredent, double myCredits){ }
    public double Transaction
    (string myCredentials, string shareName, Option whatToDo, int howMany) { }
    public delegate void ShareChanged(object src, ShareChange sc);
    public event ShareChanged OnShareChanged;
}
```

- **We can now implement this complex class as a mock and open a backdoor**



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
46

Do it Yourself – Example (4)

- **Methods to initialize mock / Methods to trigger events**

```
public void InitializeShares(Share[] arg) {
    foreach (Share s in arg)
    {
        _shares.Add(s.Name, s);
    }
}

public void TriggerEvent(string nm, DateTime dt,
                        double ov, double nv) {
    if ((OnShareChanged != null) &&
        _shares.ContainsKey(nm))
        OnShareChanged(this, new ShareChange(nm, dt, ov, nv));
}
```



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
47

Do it Yourself – Example (5)

- **The unit test could then be as follows:**

```
ShareNotificationServer srv = new ShareNotificationServer();
ShareClient sc = new ShareClient(srv, "Ms", 100000);
ShareNotificationServer.Share s1 =
    new ShareNotificationServer.Share("MSFT", 80);
ShareNotificationServer.Share s2 =
    new ShareNotificationServer.Share("IBM", 100);
ShareNotificationServer.Share s3 =
    new ShareNotificationServer.Share("INTEL", 50);
srv.InitializeShares(
    new ShareNotificationServer.Share[] {s1, s2, s3});
sc.AddRule(Option.BUY, "MSFT", 100, 65);
srv.TriggerEvent("MSFT", DateTime.Now, 80, 75);
srv.TriggerEvent("MSFT", DateTime.Now, 75, 60);
// ...
```



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
48

Do it Yourself – Example (6)

- Further issues include to check the interaction between the class under test and the mock object
- For this purpose, we can include methods to set our expectations
- And a verify-Method that checks whether our expectations are conforming with the real behavior of the class under test

```
public void ExpectAndReturn
(string method, object result, params object[] args) { }
public void ExpectAndThrow
(string method, object result, params object[] args,
object exception) { }
public void Verify() {
    // check if all happened what we expected
    // if not: Assert.Fail();
}
```



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
49

Do it Yourself – Example (7)

- In the unit test we could then write at start up:

```
srv. ExpectAndReturn("RegisterClient", null, „Ms“, 100000);
srv. ExpectAndReturn(„Transaction“, 94000, „Ms“, „MSFT“,
Option.BUY, 80);
// more and more ...
// And finally:
srv. Verify();
```

- If our expectations are not met, the verify-method will make the unit test fail
- This manual approach is tedious and error-prone, even in simple cases
- Thus, we would like to have tool support



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
50

DotNetMock

- Although there is Mock support in NUnit we will use the more comprehensive DotNetMock framework
- For demonstrating this framework I will resort to a much simpler example: a traffic light controller
- The controller itself is what we are going to implement but the traffic light class is not available yet
- Thus we introduce the interface of our real traffic light and provide a mock implementation



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
51

DotNetMock Example (1)

- This is the interface supposed to be implemented by the real object and our mock

```
public enum State { Red, Yellow, Green, OutOfOrder,
    PermanentRed, PermanentGreen, PermanentYellow };
public class OutOfOrderException : Exception { }
public interface ITrafficLight
{
    State Current { get; }
    void PowerOn();
    void PowerOff();
    void NextState();
    void AlwaysGo();
    void AlwaysStop();
    void Blink();
    void SetIdent(string id);
}
```



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
52

DotNetMock Example (2)

- Our traffic controller looks like this:

```
public class TrafficLightController {
    protected ITrafficLight _iTL;
    public TrafficLightController(ITrafficLight TL) {
        _iTL = TL; _iTL.SetIdent("Light4711");
    }
    public void PowerOn() {
        _iTL.PowerOn();
    }
    public void NextState() {
        _iTL.NextState();
    }

    public void SetPermanent(State s) {
        if (s == State.Red) _iTL.AlwaysStop();
        else if (s == State.Green) _iTL.AlwaysGo();
        else _iTL.Blink();
        else throw new ArgumentException();
    }
    public void PowerOff() {
        _iTL.PowerOff();
    }
    public State Current { get { return _iTL.Current; } }
}
```

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
53



Software &
Engineering
Architecture

DotNetMock Example (3)

- Now we are going to provide the Mock Object. We just focus for now on the method where the identity of the traffic light is set:

```
using TrafficControl;
using DotNetMock;

namespace MockTrafficLightLibrary
{
    public class MockTrafficLight : MockObject, ITrafficLight
    {
        ExpectationValue _ident = new ExpectationValue("ident");
        public string ExpectedIdent
        {
            set { _ident.Expected = value; }
        }
        public void SetIdent(string id)
        {
            _ident.Actual = id;
        }
        // everything else omitted
    }
}
```

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
54



Software &
Engineering
Architecture

DotNetMock Example (4)

- Finally we are able to develop the unit test:

```
using NUnit.Framework;
using TrafficControl;
using MockTrafficLightLibrary;

namespace TrafficLightTestUnitLibrary
{
    [TestFixture]
    public class TestTrafficLight {
        [Test]
        public void TestSetIdent() {
            MockTrafficLight mTL
                = new MockTrafficLightLibrary.MockTrafficLight();
            mTL.ExpectedIdent = "Light4711";
            TrafficLightController tlc // here we expect SetIdent
                = new TrafficLightController(mTL);
            mTL.Verify();
        } // everything else omitted
    }
}
```

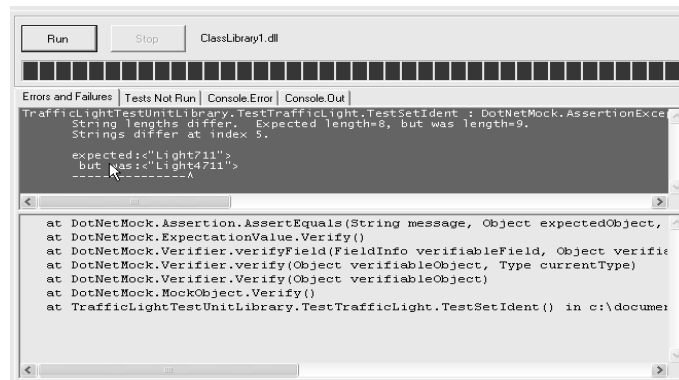


Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
55

DotNetMock Example (5)

- I have inserted an error to trigger an assertion failure caused by verify:



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
56

Dynamic Mocks

- The static approach used by DotNetMock is very helpful due to the different helper methods included
- For really big interfaces of unavailable environment objects even this approach is far too time-consuming
- Often the class under test does only access a small subset of the environment objects
- You definitely don't want to deal with methods your class under test doesn't require
- In these contexts a dynamic and reflective approach is superior
- Dynamic Mocks are exactly dealing with these issues



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
57

Dynamic Mocks Example (1)

- Dynamic Mocks require an interface to implement. Here we introduce an interface to externalize objects
- All objects that implement ICanPersist are serializable

```
namespace Persistence
{
    public interface ICanPersist { }
    public interface IPersist
    {
        void SetFileName(string name);
        void Store(object os);
        object Load();
        DateTime LastStored { get; }
    }
}
```



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
58



Dynamic Mocks Example (2)

- The class under test is responsible to handle storing/loading of objects with the help of the IPersist implementation

```
public class PersistenceManager {
    // lot of stuff omitted
    // new object persistence creation:
    public PersistenceManager
        (object objectToManage, IPersist db, string fileName) {
        if (!(objectToManage is IPersist))
            throw new ArgumentException();
        _objectToManage = objectToManage;
        _provider = db;
        _provider.SetFileName(fileName);
        Store();
    }
    public void Load() {
        object tmp = _provider.Load();
        Modify(tmp);
    }
    public void Store() ...
    public void Modify(object newValue) ...
    public DateTime LastStored() ...
}
```

20.01.2005
59

Dynamic Mocks Example (3)

- And now we specify the behavior of our dynamic mock as well as its expected values and calls in the unit tests:

```
[Test]
public void TestInitAndChange() {
    DynamicMock mock = new DynamicMock(typeof(IPersist));
    IPersist iPers = (IPersist)mock.Object;
    HelperObject ho1 = new HelperObject("A");
    HelperObject ho2 = new HelperObject("B");
    mock.Expect("SetFileName", "tempfile");
    mock.SetValue("LastStored", new DateTime(2005, 1, 1));
    mock.Expect("Store", ho2);
    PersistenceManager pm
        = new PersistenceManager(ho1, iPers, "tempfile");
    DateTime d1 = pm.LastStored();
    pm.Modify(ho2);
    pm.Store();
    mock.SetValue("LastStored", new DateTime(2005, 1, 2));
    DateTime d2 = pm.LastStored();
    Assert.IsTrue(d1.CompareTo(d2) < 0);
    mock.Verify();
}
```

20.01.2005
60

DotNetMock – Pre-built Mocks

- **There are some predefined (general) mock objects available in DotNetMock.Framework**
 - Data
 - Security
 - ComponentModel
- **Hopefully, there will be whole mock libraries in upcoming years**



Software &
Engineering
Architecture

Test your Design - Design your Test

- **Unit tests frameworks are great tools but they don't tell you what to test and how to test**
- **In addition, how do good tests look like?**
- **There is a lot of excellent literature on these topics**
- **In this talk I'll steal some of the relevant stuff from books like**
 - JUnit Pocket Guide (Kent Beck)
 - Pragmatic Unit Testing in C# (Andrew Hunt, David Thomas)
 - Unit Tests mit Java (Johannes Link)



Software &
Engineering
Architecture

Properties of Good Tests (1)

- **According to Hunt/Thomas a good test should reveal A-TRIP properties**
- **Automatic:**
 - Unit tests should run automatically (test invocation and result check)
 - Press of GUI button or execution of a command
 - No user interaction required
 - In the optimal case a separate machine is running all the tests checked in in the background e.g. by continuous integration with Cruise Control
 - Test should manage itself whether it passed or failed without requiring users



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
63

Properties of Good Tests (2)

- **Thorough:**
 - Test everything that could possibly break
 - But only as detailed as necessary (not necessarily every line). Bugs often occur in some hot spot areas
 - You may use tools such as NCover for test coverage analysis
- **Repeatable**
 - Test should always produce the same results even if the order is changed
 - Don't depend on anything external in the test that isn't under your control
 - Use mock objects to isolate the item under test from environment issues
 - If tests are not repeatable they might be worthless



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
64

Properties of Good Tests (3)

- **Independent**

- Tests shouldn't depend on each other or on their environment
- Test only one aspect at a time, maybe a feature provided by a single or small set of production methods
- Sometimes however, you'll need a set of test methods to exercise one single feature of the production code
- Don't depend on other tests or on the order of test execution

- **Professional**

- Unit test code must follow the same quality requirements as production code
- Don't test trivial things that aren't helpful such as simple property accessors
- Expect that there is almost the same size of test code as production code



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
65

What To Test

- **If we are building new production code what could we test?**

- **Hunt/Thomas recommend Right-BICEP**

- Right
- Boundary Conditions
- Inverse Relationships
- Cross-Check
- Error Conditions
- Performance

- **Right Results:**

- Validation of results is used to check conformance with expectation
- Ask yourself: If the code ran correctly, how would I know?



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
66

What to Test - Boundary Conditions (B-1)

- **Boundary conditions are the most valuable part of unit testing**
- **Examples:**
 - Duplicates in lists that shouldn't have duplicates
 - Empty or missing values
 - Badly formatted data
 - Bogus input values (such as „Ab7C €“)
 - Ordered lists that aren't ordered, and vice versa
 - Out of order events (e.g., open before read)
- **Use CORRECT approach: Conformance, Ordering, Range, Reference, Existence, Cardinality, Time**



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
67

What to Test - Boundary Conditions (B-2)

- **Conformance**
 - Whenever you expect data to be in a specific format determine would could get wrong, e.g.:
 - phone numbers
 - currencies
 - credit card numbers
 - XML schemas
 - record format of a file
 - E-mails, URLs
 - How is the production code supposed to handle conformance violations?



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
68



What to Test - Boundary Conditions (B-3)

• Ordering

- Position of data in larger collection can be relevant, especially beginning or end
- Input order may be different from order: e.g. consider log messages that are re-ordered according to their creation time, priority queues, restaurant orders, purchase orders where shipping of items might happen at different times
- Examples:
 - What if sort algorithm receives an already ordered list or what if the list is ordered in reverse order?
 - If production code depends on ordering check that items are correctly ordered



What to Test - Boundary Conditions (B-4)

• Range

- Test how the system under test deals with range violations
- Application-domain specific ranges:
 - Age might be not higher than specific values
 - There is no negative number of things
 - An angle might be between 0° and 360°
- Physical constraints such as overflow, empty objects
- Beware of class invariants: e.g. rectangles might have different lengths of their lines while requiring the area to be always the same
- Index violations: start and index are same, first greater than last, negative index, index greater than allowed, count not equal to number of items

What to Test - Boundary Conditions (B-5)

- **Reference**

- References to external objects on which the method depends
- State conditions within implementation class (e.g. user must be logged on to perform method)
- Test your production code how it deals with situations where some preconditions are violated (e.g., state that doesn't allow a specific method)
- Especially important here are
 - Preconditions
 - Postconditions



Software &
Engineering
Architecture

What to Test - Boundary Conditions (B-6)

- **Existence**

- Test what happens if some things don't exist (non-existing file or data, null, 0, „“, ...)
- Example:
 - Method returns null because it can't find a specific value
 - Method expects a valid connection but there is an error
 - Calculation returns 0

- **Cardinality**

- Check for 0, 1, more than 1, minimum, maximum
- Beware of fence-post problem



Software &
Engineering
Architecture

What to Test – Boundary Conditions (B-7)

- **Time**

- Test timing issues (a little bit related to ordering)
 - Relative time: e.g., open before read before close, timing and timeout issues to prevent waiting forever
 - Absolute time: e.g. things that must happen at specific times
 - Concurrency: how to deal with synchronization issues



Software &
Engineering
Architecture

What to Test – I C and E

- **Inverse relationships**

- can be tested by applying the opposite operation, e.g., `my_sqrt(n) * my_sqrt(n) == n`
- but use the reverse operation from another source

- **Cross-Check Using Other Means**

- Use alternative ways to check your result: `Math.sqrt(n) == my_sqrt(n)`

- **Error Conditions**

- Check how your code behaves in error situations such as network errors, file access problems
- Consider using mock objects to trigger those errors
- Examples: running out of memory or disk space, issues with time, network errors, system load, color palette, high and low video resolution



Software &
Engineering
Architecture

What to Test - Performance

- Test all performance characteristics of your application
- What about growing input sizes
- Use timers to measure timing behavior
- Performance tests might take long to run. Thus, consider to execute them less often than all the other unit tests



Software &
Engineering
Architecture

Test First Reloaded

- In the initial slides I talked about Test-First
- After we've learned about Unit Testing we are ready to understand and use the Test First approach
- It basically comprises the following steps:
 - Before writing any line of production code introduce a small test
 - Only write as much production code as necessary to make the test succeed
 - In further iterations (not more than 20 minutes for each) add test code first and then production code to make the extended test succeed
 - Before adding production code you might refactor the solution
 - When going to integrate your code in the complete system all unit tests must successfully execute



Software &
Engineering
Architecture

Test First Example (1)

- Kent Beck offers a very simple example to illustrate the idea
- Suppose, you are going to calculate factorial
- Tests should check for ≤ 0 , 1, and more
- Here is our first test (we expect failure for all negative values):

```
using NUnit.Framework;

[TestFixture]
public class UnitTestFactorial
{
    [Test, ExpectedException(typeof(ArgumentException))]
    public void TestNegative()
    {
        MathOperations.Factorial(-1);
    }
}
```

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
77



Software &
Engineering
Architecture

Test First Example (2)

- To adhere to the specification we just implement the necessary code:

```
public class MathOperations {
    public static int Factorial(int arg) {
        if (arg < 0) throw new ArgumentException();
        return 0;
    }
}
```

- Obviously the test succeeds. The next requirement is that factorial for 0 should be 0 and for 1 should yield 1:

```
[Test]
public void Test0And1() {
    Assert.AreEqual(1, MathOperations.Factorial(0));
    Assert.AreEqual(1, MathOperations.Factorial(1));
}
```

- This test is initially failing. We must refactor the production code



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
78

Test First Example (3)

- **The result of our refactoring is (the simplest solution that could possibly work):**

```
public class MathOperations {
    public static int Factorial(int arg) {
        if (arg < 0) throw new ArgumentException();
        return 1;
    }
}
```

- **In the last step we want to test many which also fails:**

```
[Test]
public void TestMoreThanOne() {
    Assert.AreEqual(2, MathOperations.Factorial(2));
    Assert.AreEqual(6 * MathOperations.Factorial(5),
        MathOperations.Factorial(6));
    Assert.AreEqual(10 * MathOperations.Factorial(9),
        MathOperations.Factorial(10));
}
```



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
79

Test First Example (4)

- **The only possibility to refactor now is:**

```
public class MathOperations {
    public static int Factorial(int arg) {
        if (arg < 0) throw new ArgumentException();
        if ((arg == 0) || (arg == 1))
            return 1;
        else
            return arg * Factorial(arg - 1);
    }
}
```

- **And now all tests succeed ☺**



Software &
Engineering
Architecture

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
80

Unit Test Frameworks for .NET

- **There are different frameworks available:**

- NUnit is one of the best because of its idiomatic approach:
<http://www.nunit.org/>
- mbUnit <http://mbunit.tigris.org/> is bundled with testdriven.net
(<http://www.testdriven.net>)
- csUnit (<http://www.csunit.org/index.php>)
- More resources and information available on
<http://www.xprogramming.com/>
- General patterns for unit testing and building xUnit frameworks:
<http://tap.testautomationpatterns.com:8080/index.html>



Software &
Engineering
Architecture

Mock Frameworks for .NET

- The framework I've introduced in my talk was DotNetMock:
<http://sourceforge.net/projects/dotnetmock/>
- In NUnit you'll also find a mock library
- NMock: <http://www.nmock.org/>. See also article
<http://msdn.microsoft.com/msdnmag/issues/04/10/NMock/default.aspx>
- General information available on <http://www.mockobjects.com>



Software &
Engineering
Architecture



Software &
Engineering
Architecture

Related Tools

- **Unit test tools (frameworks for unit test and mocks) are one part of the story**
- **Unit testing should be integrated with**
 - a tool to check test coverage such as nCover
 - a tool that checks adherence to conventions and rules such as FxCop
 - a tool that helps organizing the build such as NAnt
 - a tool for continuous integration such as CruiseControl.NET
- **One point is important: you should organize Unit tests in a parallel branch of your repository / file system**

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
83



Software &
Engineering
Architecture

Books

- **Kent Beck: Test Driven Development: By Example, Addison Wesley, 2002**
- **This book is on JUnit but nonetheless recommendable for .NET developers: Kent Beck, JUnit Pocket Guide, O'Reilly, 2004**
- **Bill Hamilton, NUnit Pocket Reference, O'Reilly, 2004**
- **Johannes Link, Unit Tests mit Java, dpunkt.verlag, 2002**
- **Andrew Hunt, David Thomas, Pragmatic Unit Testing In C# with NUnit, The Pragmatic Programmers, LLC, 2004 – also available as PDF**
- **Neil Roodyn, eXtreme .NET : Introducing eXtreme Programming Techniques to .NET Developers (Microsoft .Net Development), Microsoft Press, 2004**

© Siemens AG, CT SE 2, Michael Stal,
20.01.2005
84

Summary

- **Unit Test Frameworks** such as NUnit support developers to build high quality code and obtain high confidence
- **Unit tests** also document what you write
- They help to make sure nothing is broken when new features are added to a system
- **Unit testing is very easy** as everything is automated
- **Use Mock objects** to simulate those parts of the environment which cannot be provided physically
- **Unit tests are not restricted to agile processes** such as Scrum or XP. Agile processes need unit testing and refactoring to work
- **Unit testing requires some additional efforts** that definitely pays back during development
- **Get addicted to tests.** Or as Erich Gamma said „Get Test-infected!“



Software &
Engineering
Architecture