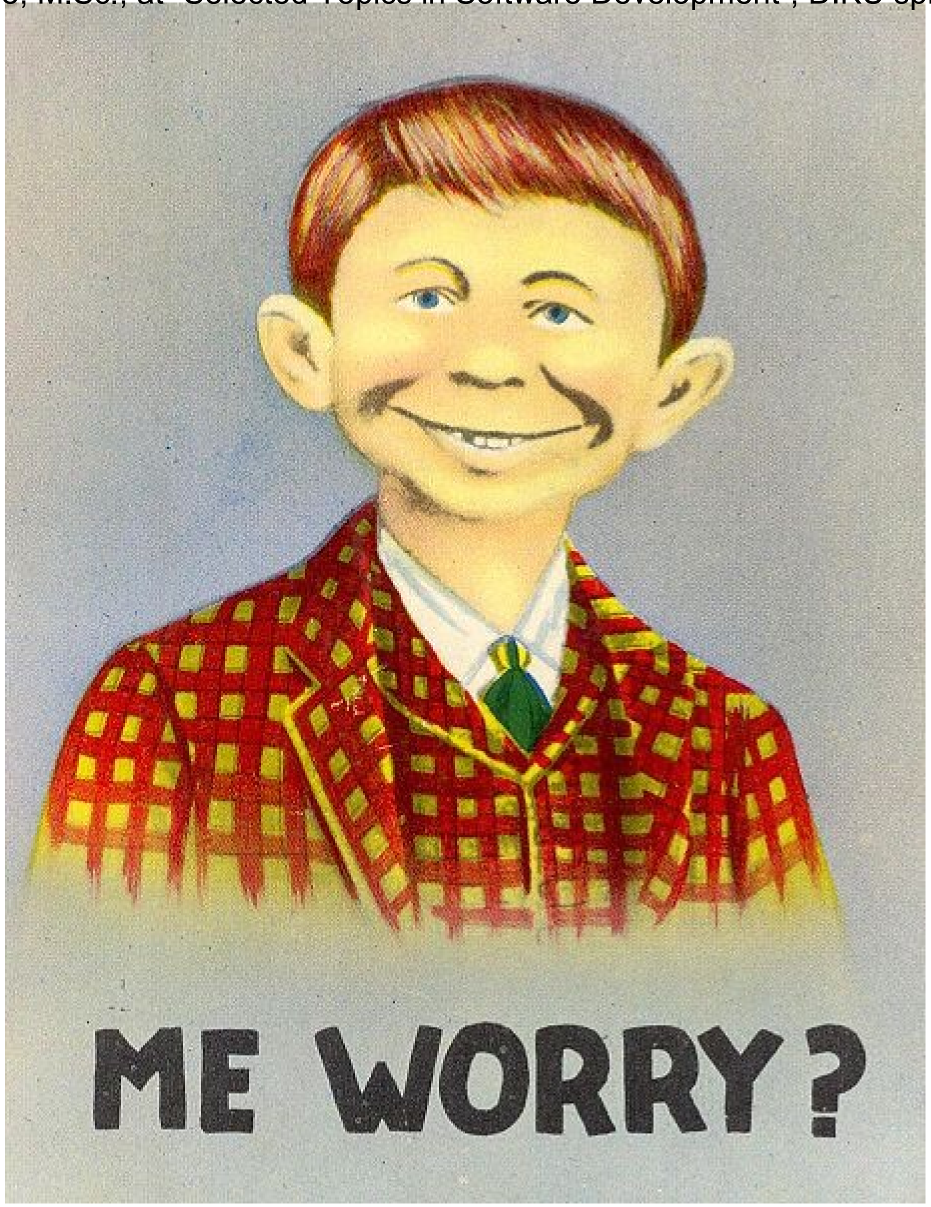
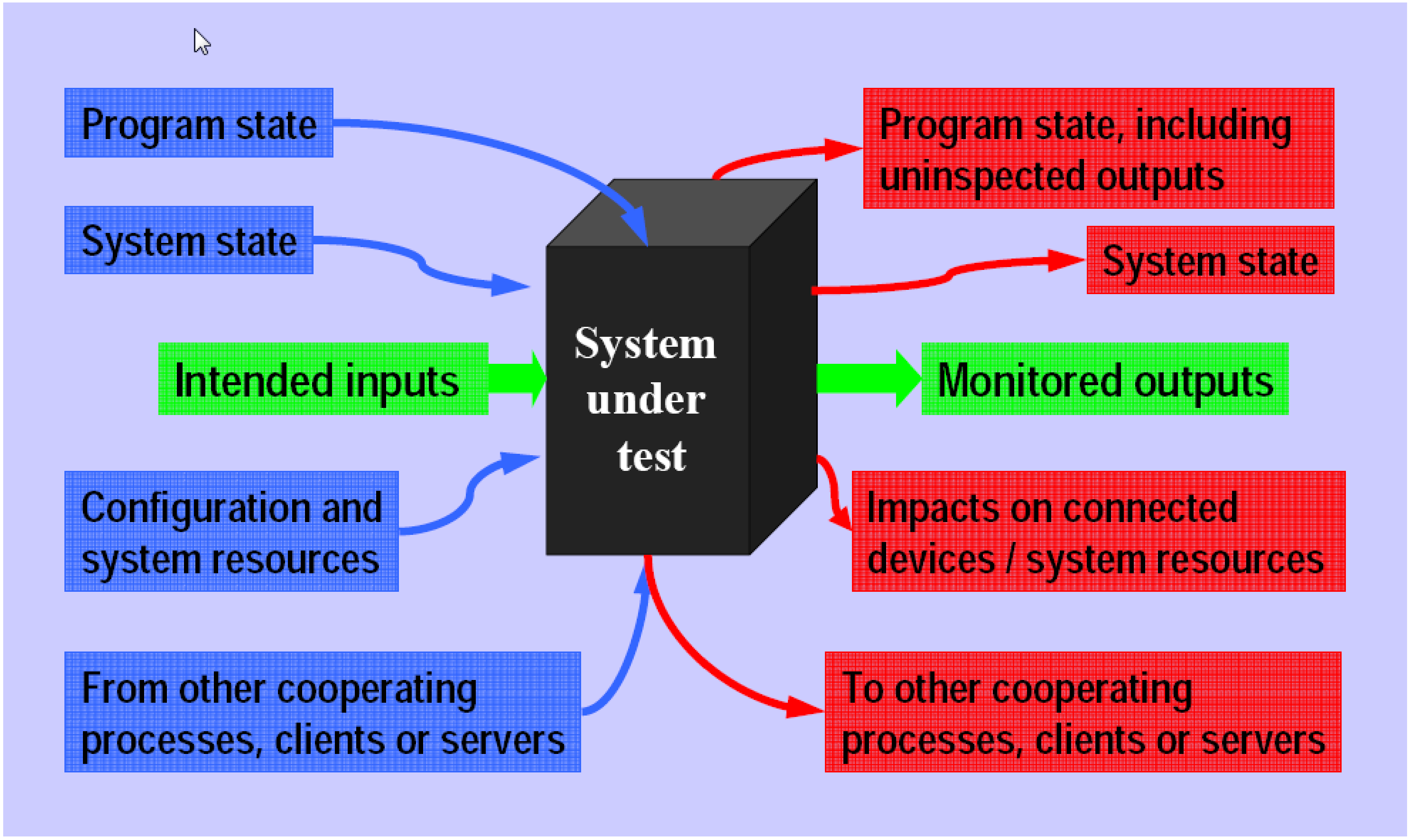
Software Testing Concepts and Tools

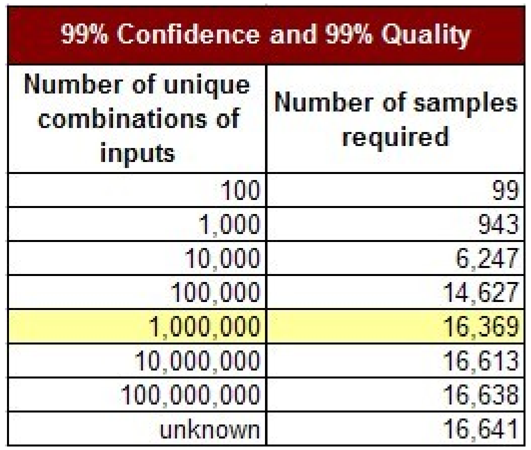
Presented by Lars Yde, M.Sc., at ”Selected Topics in Software Development”, DIKU spring semester 2008



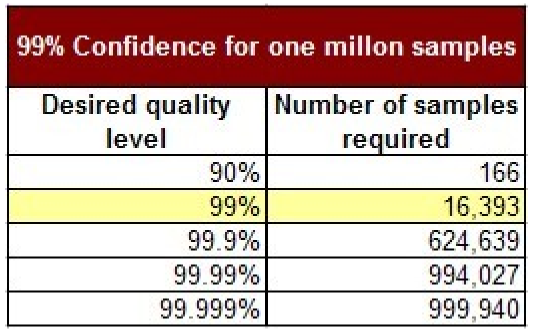
The what's and wherefore's

- To test is to doubt. So why do we doubt ?
  - If you're any good, why not just write proper=correct code ? Specs, typos, and 9 levels of complexity
  - Even with perfect code, we cannot control all outside variables
- The scope of the problem
  - Binder, chap. 1: A Small Challenge, 65 permutations
  - Rollison, SW Test & Perf, p.33: *”Using only the characters 'A' - 'Z' the total number of possible character combinations using for a filename with an 8-letter filename and a 3-letter extension is 26^8 + 26^3, or 208,827,099,728”*
  - The virtue of the pragmatist approach


Program state
System state
Intended inputs
Configuration and system resources
From other cooperating processes, clients or servers
System under test
Program state, including uninspected outputs
System state
Monitored outputs
Impacts on connected devices / system resources
To other cooperating processes, clients or servers



99% Confidence and 99% Quality	
Number of unique combinations of inputs	Number of samples required
100	99
1,000	943
10,000	6,247
100,000	14,627
1,000,000	16,369
10,000,000	16,613
100,000,000	16,638
unknown	16,641



99% Confidence for one millon samples	
Desired quality level	Number of samples required
90%	166
99%	16,393
99.9%	624,639
99.99%	994,027
99.999%	999,940

Verification cost is asymptotic to the desired quality level.

Graphics courtesy Scott Sehlhorst

Systemic levels of testing

- Unit
  - Black box, white box, grey box
- Integration
  - Shared memory, shared interfaces, shared messages
  - Load testing, performance testing, regression testing
- System: full monty, in realistic ”ecosystem”
- Installation / deployment
  - Precondition/postcondition, load, regression
- Acceptance testing
  - Manual (e.g. GUI), automated (e.g. Formal ver.)

Testing schools of thought

- Formal approaches
  - Binder, chap.6, combinatorial: enumerate, cull, verify, impractical, humbled by complexity
  - Binder, chap.7, state machines: flow visualization, coherence (p.238), useful as cscw artifact
  - Binder, chap.9-15, patterns: templates for test design, good craftsmanship, tools of the trade, heuristics in a combinatorial explosion. Recognizing the facts.
- Agile approach
  - Test often, test repeatedly, test automatically

TDD testing - an agile approach

- Simple recipe
  - Design as required
  - Write some test cases
  - See that they fail (no code yet to exercise)
  - Make them pass by writing compliant code
  - Goto start
- Write iteratively, code and design iteratively
- Gradual improvement through design and selection – evolving design + code

TDD testing

- Automated tests that are
  - Concise
  - Self checking
  - Repeatable
  - Robust
  - Sufficient
  - Necessary
  - Clear
  - Efficient
  - Specific
  - Independent
  - Maintainable
  - Traceable
- Also room for automated acceptance testing

TDD challenges / headaches

- Sealed interfaces in legacy code
- Multithreaded designs with intricate lock patterns
- Deeply tiered implementations with no separation
- Automated GUI testing (allow for scripting, please)
- Mismanagement, business decisions
- Database dependencies (consider mock objects)
- Overly long and complex test cases – easier to crank out than you might think...
- Bugs in testing harnesses and frameworks

Pragmatic testing

- Experiment is part & parcel of the scientific method
  - Test to falsify – not to validate. Prove false, not true
  - Prove beyond a reasonable doubt. Good enough for hanging, good enough for testing.
  - Strategic levels of testing (not systemic levels)
    - Test exhaustively
    - Test exploratorily
    - Test sporadically
    - Test stochastically
    - Roll your own

POLICE LINE DO NOT CROSS
POLICE LINE DO NOT CROSS
POLICE LINE DO NOT CROSS
BLOOD SPECIMEN
BLOOD SPECIMEN
BLOOD SPECIMEN

Tactics for targeted testing

- Equivalence classes – formal and heuristic
- Boundary values – peaks and valleys rather than flatlands
- Risk analysis – unclear, complex, legacy, ”here be dragons”
- Use coverage tools – which portions of code exercised: Ncover
- Experience and intuition

Tools

- Unit testing frameworks + extras
  - Junit
  - Nunit / Ncover / Nant (versatile, GUI, convenient, extendible)
  - Boost (organise into cases, suites, extensive assert library, simple and effective)
  - CppUnit (extending framework base classes, assert library, clumsy)
- Example (Nunit) – assume class Account

```
•  namespace bank
•  {
•    public class Account
•    {
•      private float balance;
•      public void Deposit(float amount)
•      {
•        balance+=amount;
•      }

•      public void Withdraw(float amount)
```

Nunit testing cont'd

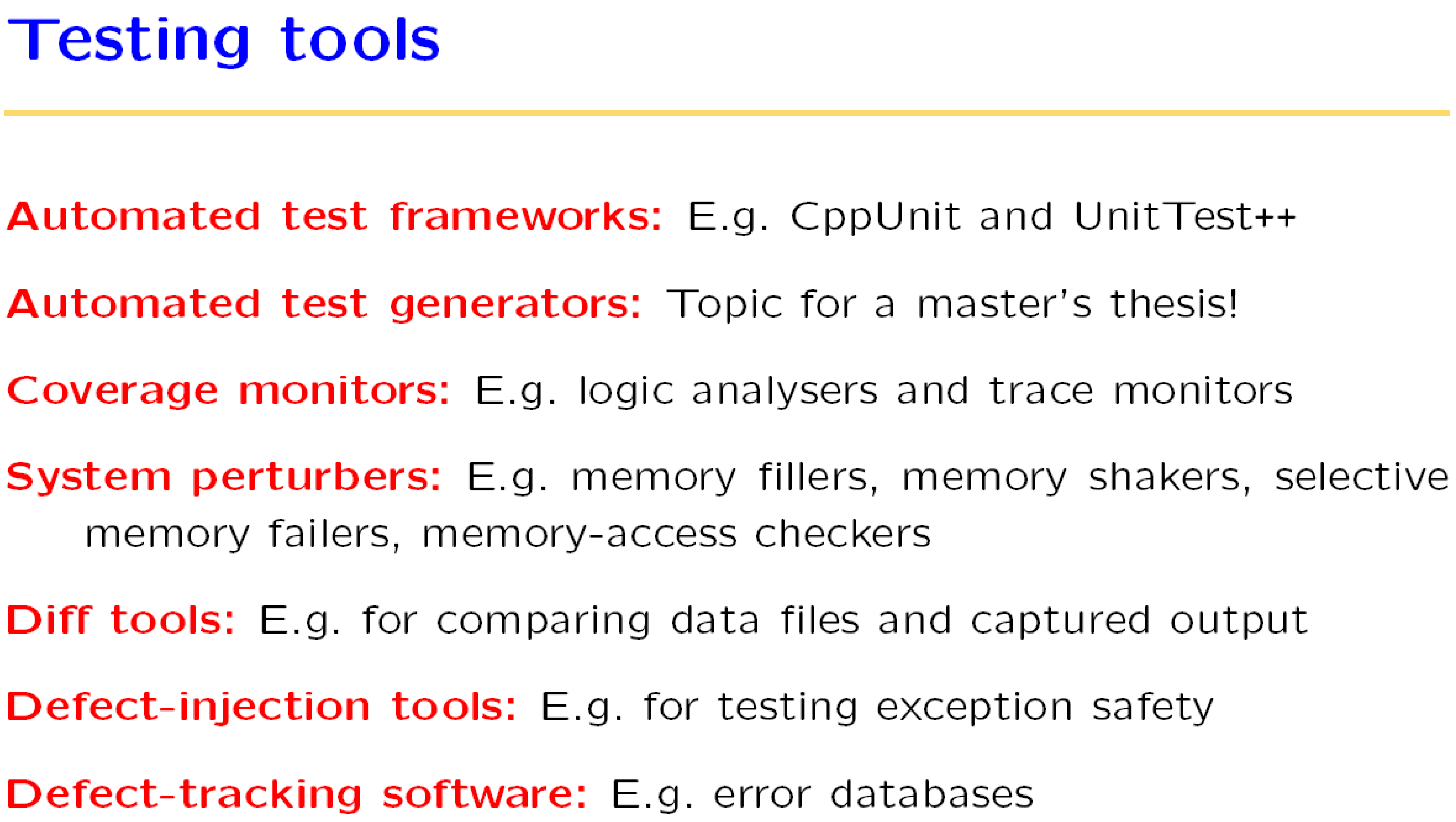
```
namespace bank
{
  using System;
  using NUnit.Framework;

  [TestFixture]
  public class AccountTest
  {
    Account source;
    Account destination;

    [SetUp]
    public void Init()
    {
      source = new Account();
      source.Deposit(200.00F);
      destination = new Account();
      destination.Deposit(150.00F);
    }

    [Test]
    public void TransferFunds()
    {
      source.TransferFunds(destination,
100.00f);
      Assert.AreEqual(250.00F,
destination.Balance);
      Assert.AreEqual(100.00F,
source.Balance);
    }
```

Jyrki Slide on Testing Tools



Testing tools

Automated test frameworks: E.g. CppUnit and UnitTest++

Automated test generators: Topic for a master's thesis!

Coverage monitors: E.g. logic analysers and trace monitors

System perturbers: E.g. memory fillers, memory shakers, selective memory failers, memory-access checkers

Diff tools: E.g. for comparing data files and captured output

Defect-injection tools: E.g. for testing exception safety

Defect-tracking software: E.g. error databases

Jyrki slide on testing tools

Specific tools relevant for your assignment

IDE: Eclipse, `emacs`

Scripting: Python, `make`

Shell commands: `diff`, `grep`, `find`

Version control: `cvs` (includes diff, merge, and log facilities)

Concept correctness: Boost concept archetypes

Debugging: `ddd`

Memory-usage monitoring and profiling: `valgrind`

Benchmarking: `benz`

ANKHSvn test case – studio plugin

```
// $Id$
#pragma once
#include "stdafx.h"
using namespace NUnit::Framework;

namespace NSvn
{
    namespace Core
    {
        namespace Tests
        {
            namespace MCpp
            {
                [TestFixture]
                public __gc class ManagedPointerTest
                {
                public:
                    /// <summary>Test that the thing works</summary>
                    [Test]
                    void TestBasic();

                    [Test]
                    void TestAssignment();

                    [Test]
                    void TestCopying();

                };
            }
        }
    }
}
```

ANKHSvn test case – studio plugin

```
// $Id$
#include "StdAfx.h"
#include "../NSvn.Core/ManagedPointer.h"
#include "managedpointertest.h"
#using <mscorlib.dll>
using namespace System;
namespace
{
    void voidFunc( void* ptr )
    {
        String* string = *(static_cast<NSvn::Core::ManagedPointer<String*>* >( ptr ) );
        Assert::AreEqual( string, S"Moo world" );
    }
}

void NSvn::Core::Tests::MCpp::ManagedPointerTest::TestBasic()
{
    String* str = S"Moo world";
    ManagedPointer<String*> ptr( str );

    //check that the implicit conversion works
    Assert::AreEqual( ptr, S"Moo world" );

    //implicitly convert to void*
    voidFunc( ptr );
}

void NSvn::Core::Tests::MCpp::ManagedPointerTest::TestAssignment()
{
    String* str = S"Moo world";
    ManagedPointer<String*> ptr1( str );
    ManagedPointer<String*> ptr2( S"Bleh" );

    ptr2 = ptr1;
    Assert::AreEqual( ptr1, ptr2 );
    Assert::AreEqual( ptr2, S"Moo world" );
}
void NSvn::Core::Tests::MCpp::ManagedPointerTest::TestCopying()
{
    String* str = S"Moo world";
    ManagedPointer<String*> ptr1( str );
    ManagedPointer<String*> ptr2( ptr1 );

    Assert::AreEqual( ptr1, ptr2 );
```

Exhausting your Test Options – Doug Hoffmann

We designed a test where each of the 64K CPUs started with a different value and tested each value 65,536 greater than the last one. Then, each CPU would ask a neighbor to compute the square root differently to check the original result.

Two Errors in Four Billion

Two hours later, we made the first run using our biggest processor configuration. The test and verification ran in about six minutes or so, but we were confused a bit when the test reported two values in error. Both the developer and I scratched our heads, because the two values didn t appear to be special in any way or related to one another. Two errors in four billion. The developer shook his head and said, No& This is impossi& Oh!& Okay? He thumbed through the microcode source listing and stabbed his finger down on an instruction. Yeah there it is! There was only one thing that could cause those two values to be wrong, and a moment s thought was all it took to discover what that was!

He gleefully tried to explain that the sign on a logical micro instruction shift instruction was incorrect for a particular bit pattern on a particular processor node at 11 P.M. on the night of a full moon (or some such it didn t make a lot of sense to me at the time but we caught, reported, and fixed the defect, fair and square). We submitted the defect report, noting only the two actual and expected values that miscompared and attaching the test program. Then, in just a few minutes, he fixed the code and reran the test, this time with no reported errors. After that, the developers ran the test before each submission, so the test group didn t ever see any subsequent problems.

I later asked the developer if he thought the inspection we had originally planned would likely have found the error. After thinking about it for a minute, he shook his head. I doubt if we d have ever caught it. The values weren t obvious powers of two or boundaries or anything special. We would have looked at the instructions and never noticed the subtle error.

Failures can lurk in incredibly obscure places. Sometimes the only way to catch them is through exhaustive testing. The good news was, we had been able to exhaustively test the 32-bit square root function (although we checked only the expected result, not other possible errors like leaving the wrong values in micro registers or not responding correctly to interrupts). The bad news was that there was also a 64-bit version, with four billion times as many values that would take about 50,000 years to test exhaustively.

Testing patterns - example

Name: Architecture Achilles Heel Analysis

Elisabeth Hendrickson, Grant Larsen

Objective: Identify areas for bug hunting – relative to the architecture

Problem: How do you use the architecture to identify areas for bug hunting?

Forces:

· You may not be able to answer all your own questions about the architecture.

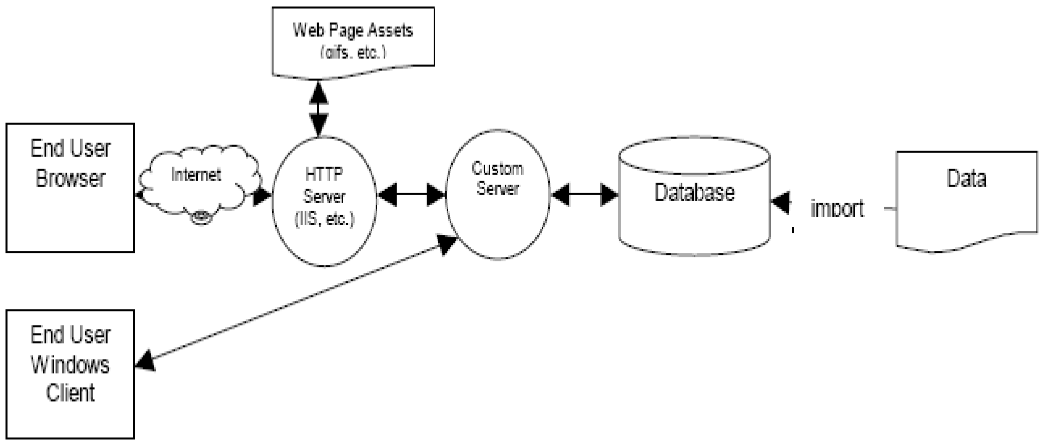
· Getting information about the architecture or expected results may require additional effort.

· The goals of the architecture may not have been articulated or may be unknown.

Solution:

An architectural diagram is usually a collection of elements of the system, including executables, data, etc., and connections between those elements. The architecture may
specify the connections between the elements in greater or lesser detail. Here's an example deployment architecture diagram. This technique works for deployment, physical, functional, or dynamic architecture maps

Testing patterns - example



In analyzing the architecture, walk through each item and connection in the diagram.
Consider tests and questions for each of the items in the diagram.

Connections

- Disconnect the network
- Change the configuration of the connection (for example, insert a firewall with
maximum paranoid settings or put the two connected elements in different NT
domains)
- Slow the connection down (28.8 modem over Internet or VPN)
- Speed the connection up.
- Does it matter if the connection breaks for a short duration or a long duration? (In
other words, are there timeout weaknesses?)

User Interface

- Input extreme amounts of data to overrun the buffer
- Insert illegal characters
- Use data with international and special characters
- Use reserved words
- Enter zero length strings
- Enter 0, negatives, and huge values wherever numbers are accepted (watch for
boundaries at 215=32768, 216=65536, and 231=2147483648).
- Make the system output huge amounts of data, special characters, reserved
words.
- Make the result of a calculation 0, negative, or huge.

Server Processes

- Stop the process while it's in the middle of reading or writing data (emulates
server down).
- Consider what happens to the clients waiting on the data?
- What happens when the server process comes back up? Are transactions rolled
back?

Databases

- Make the transaction log absurdly small & fill it up.
- Bring down the database server.
- Insert tons of data so every table has huge numbers of rows.
- In each case, watch how it affects functionality and performance on the server and
clients.

Testing patterns – a code example

Call Stack Tracer
Motivation
Assume you are using a runtime library that can throw exceptions4.
The runtime library can throw
exceptions in arbitrary functions. However, you have a multitude of
places in your code, where you call
functions of the runtime library.
In short: How do you find out, where the exception has been thrown (,
while avoiding too much "noise" in
the debug trace)?
Forces
1. You want to avoid "noise" in the debug trace.
2. The tracing facility must be easy to use.
3. You want to exactly know, where the function was called, when the
exception has been thrown.

Testing patterns – a code example

Solution

Implement a class *CallStackTracer*, that internally traces the call stack, but writes to the debug trace only, if an exception has occurred.

The mechanism is as follows: An instance of the class is bound to a block (this works for at least C++ and SmallTalk). The constructor for that instance takes as a parameter information about where the instance has been created, e.g. file name and line number.

If everything is OK, at the end of the block, the function SetComplete() is invoked on the instance of *CallStackTracer*.

When the instance goes out of scope the destructor is called. In the implementation of the constructor, the instance checks, whether the SetComplete() function was called. If not, an exception must have been thrown, so the destructor of the instance writes the information, e.g. file name, line number, etc., to the debug trace. If the SetComplete() function was called before, then the instance traces nothing thus avoid the "noise" in the debug trace.

```
class CallStackTracer {
CallStackTracer(string sourceFileName,
int sourceLineNumber) {
m_sourceFileName = sourceFileName;
m_sourceLineNumber = sourceLineNumber;
m_bComplete = false;
};
~CallStackTracer() {
if( !m_bComplete ) {
cerr << m_sourceFileName << m_sourceLineName;
}
}
void SetComplete() {
m_bComplete = true;
}
private:
string m_sourceFileName;
int m_sourceLineNumber;
};
// Some code to be covered by a CallStackTracer
void foo() {
CallStackTracer cst(__FILENAME__, __LINENUMBER__);
// all normal code goes here.
cst.SetComplete();
}
```

Note, that __FILENAME__ and __LINENUMBER__ are preprocessor macros in C++, which will be replaced during compilation by their value.