White box testing concepts

Satish Mishra

mishra@informatik.hu-berlin.de

This session

- White box testing
- Unit testing concepts for C/C++/.Net etc
  - Unit testing tools
- Testing related terms
  - Memory Leak
  - Test/Code Coverage
  - Performance Profiler
  - Dynamic test
- Brief demo of professional testing tool
  - Cantata++
- Discussions
  - Always welcome

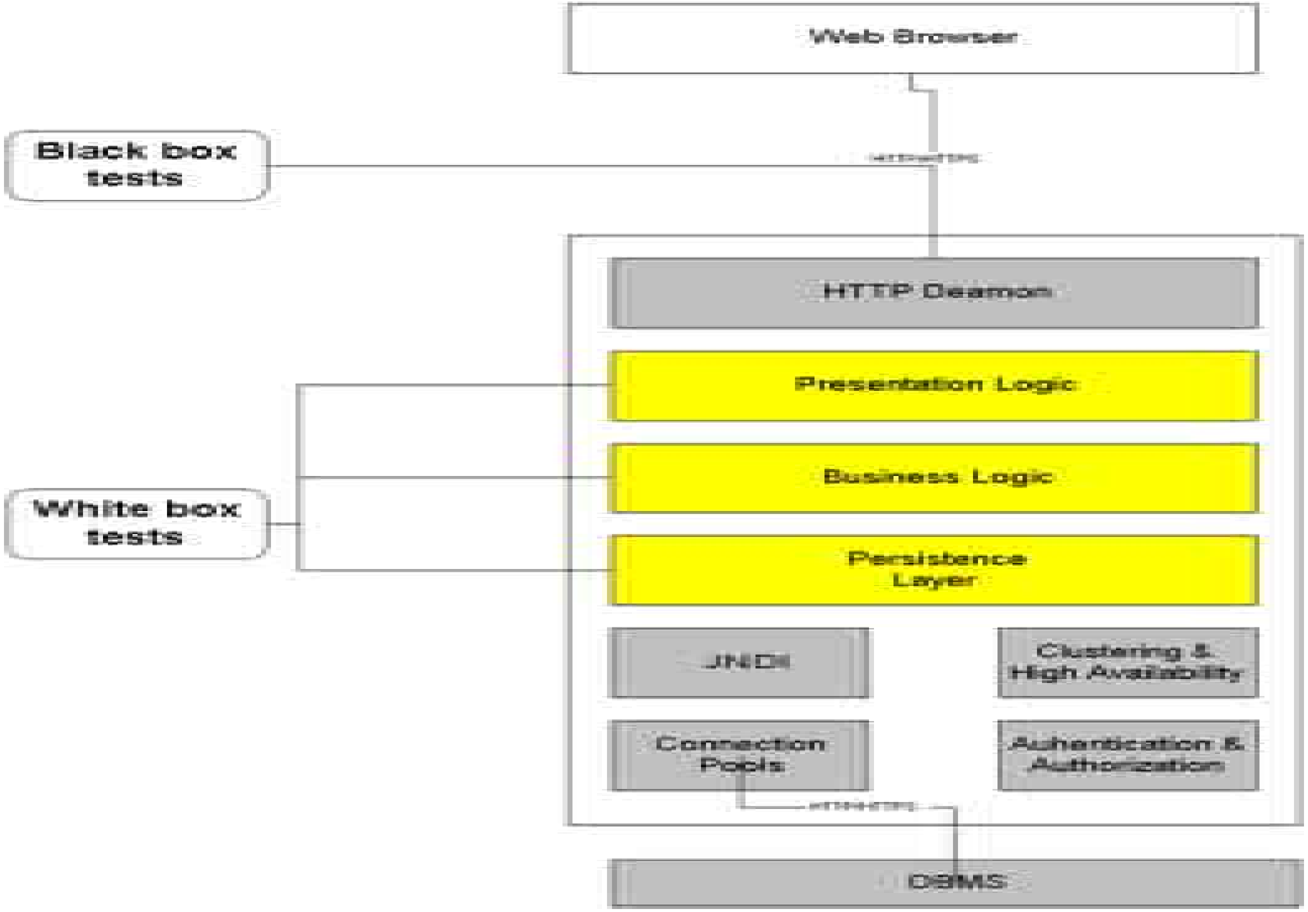
Status of assignments

- Doubts ?
- Progress ! !
- Results ??
- Suggestions

White box testing

- Also known as *glass box*, *structural*, *clear box* and *open box testing*. A software testing technique whereby explicit knowledge of the internal workings of the item being tested are used to select the test data.
- For a complete software examination, both white box and black box tests are required

White Box and Black Box tests interface to an application



Unit testing

How to test the programs of different technology ?

- C
- C++
- Microsoft technologies VB/.Net
- Web related (PHP,ASP,JSP..etc.)

How to do unit testing of above technologies ?

Unit test tools

- AdaTEST
- AQtest
- Aunit
- C++Test
- Cantata
- Check
- COMPUTE
- CppUnit
- csUnit
- CTA++
- CTB
- cUnit
- CUT
- dotunit
- EasyMock
- GrandTestAuto
- HarnessIt
- HtmlUnit
- HttpUnit
- JavaScript
- JsUnit
- JsUnit
- JTestCase
- JUnit
- JUnitEE
- JUnitX
- LingoUnit
- MinUnit
- Mock Creator
- Mock Objects
- MockMaker
- Mockry
- NUnit
- ObjcUnit
- OCUnit
- OTF - An
- PalmUnit
- PBUnit
- PerlUnit
- phpAsserUnit
- PhpUnit
- PyUnit
- QtUnit
- Ruby/Mock
- SUnit
- TagUnit
- TBGEN
- TBrun
- Test Mentor - Java Edition
- unit++
- vbUnit3 Basic
- VectorCAST
- XMLUnit
- XSLTunit

http://www.testingfaqs.org/t-unit.htm

Testing related terms

- Memory Leak
- Test/Code Coverage
- Performance Profiler
- Dynamic test

What is memory leak

- What
  - Allocating memory without releasing later
- Why bad
  - Reduce performance
  - May cause crashes
- How to solve
  - Find out where exactly memory is leaked

C/C++ memory leak

- In C/C++, it is possible to allocate space for objects (variables) dynamically during program execution. After finishing use of a dynamically allocated object, it is necessary to explicitly release the memory consumed by the object, particularly before pointers to the object go out of scope.

Memory leak example

- When a variable is created by a “usual declaration”, i.e., without **new**, memory is allocated on the “stack”.

```
{
  int i = 3;        // memory for i and obj
  MyObject obj;     // allocated on the stack
  ...
}
```

So when we delete them ??

- When the variable goes out of scope, its memory is automatically deallocated (“popped off the stack”).

```
          // i and obj go out of scope,
          // memory freed
```

Memory leak example…

- To allocate memory dynamically, we first create a pointer, e.g., **MyClass* ptr;**
- **ptr** itself is a variable on the stack. Then we create the object: **ptr = new MyClass(*constructor args*);**
- This creates the object (pointed by **ptr**) from a pool of memory called the “heap” (or “free store”).
- When the object goes out of scope, **ptr** is deleted from the stack, but the memory for the object itself remains allocated in the heap:

```
{
  MyClass* ptr = new MyClass();    // creates
object….. …
}   // ptr goes out of scope here -- memory leak!
```

Memory leak example…

To prevent the memory leak, we need to deallocate the object’s memory before it goes out of scope:

```
{
  MyClass* ptr = new MyClass();   // creates an object
  MyClass* a = new MyClass[n];    // array of objects
  ...
  delete ptr;  // deletes the object pointed to by ptr
  delete [] a; // brackets needed for array of objects
}
```

For every **new**, there should be a **delete**.
For every **new** with brackets **[],** there should be a **delete []**

Test/Code coverage

- Precondition
  - Software product under development
  - Test suite

Test / Code coverage provides a measure of how well test suite actually tests the product.

Test/Code coverage analysis

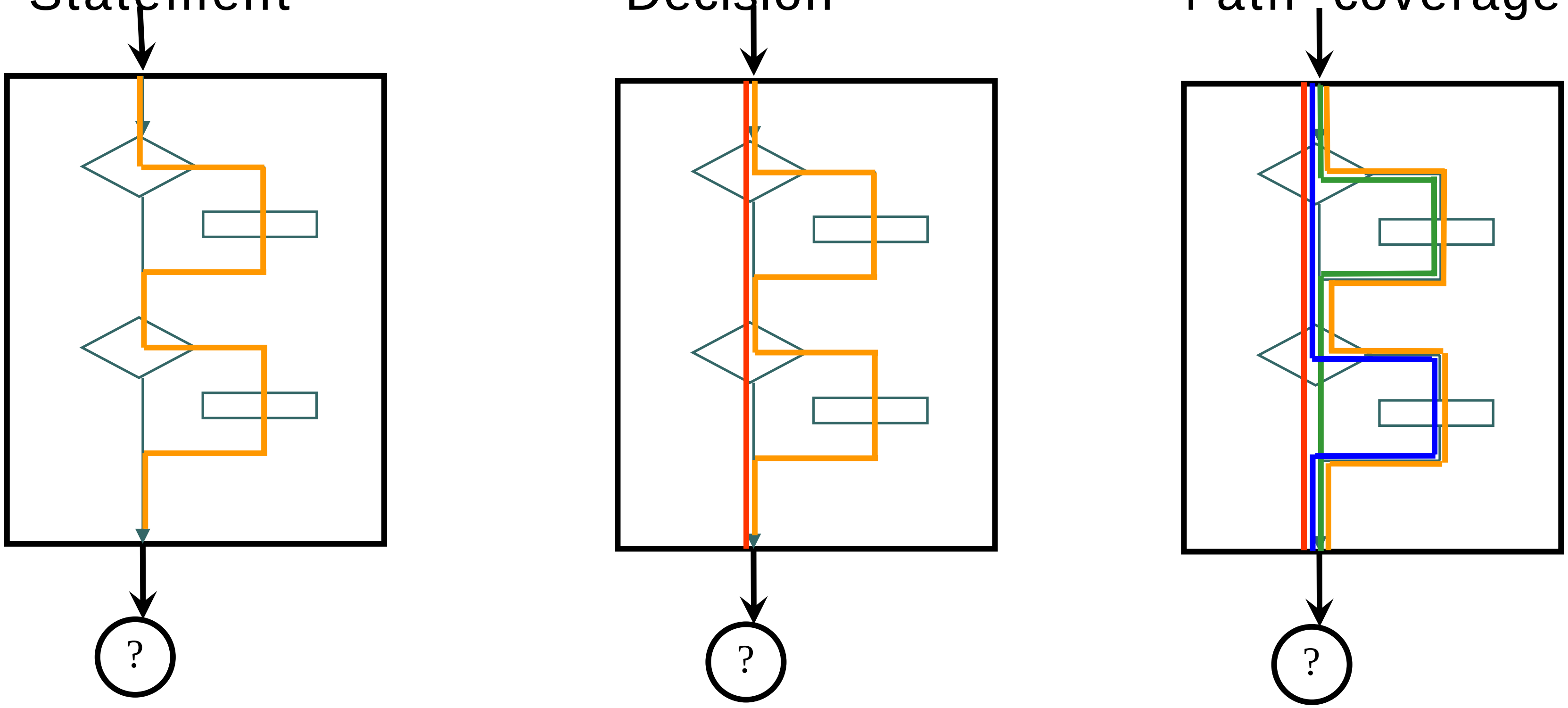
- Coverage analysis is a way of measuring how much of the code has been exercised during testing
- Coverage analysis can be used to determine when sufficient testing has been carried out
- Coverage analysis can identify unexecuted code structures
  - Add more test cases?
  - Remove dead or unwanted code!

An optional aspect is:

- Identifying redundant test cases that do not increase coverage

Test/Code coverage – examples

Statement Decision Path coverage



1 2 4 test cases

Type of coverage

- Statement coverage
- Basic block coverage
- Decision coverage
- Condition coverage
- Branch coverage
- Loop coverage

Exercise for test coverage

* We should try now..

```
#include <iostream.h>
int main(void) {
    int nDependents, Exemption;
    float Income, TaxSubTotal, TaxTotal;
    cout << "Welcome to tax calculator. Enter your yearly income: ";
    cin >> Income;

    // first if - check income
    if (Income < 0) {
            cout << "You cannot have a negative income.\n";
            return 0;
    }
    cout << "Enter the number of dependents you have, including yourself: ";
    cin >> nDependents;
    // second if - check dependents
    if (nDependents <= 0) {
            cout <<"You must have at least one dependent.\n";
            return 0;
    }
    // third if (else-if) - compute tax subtotal
    if (Income < 10000)
            TaxSubTotal = .02 * Income;
    else if (Income < 50000)
            TaxSubTotal = 200 + .03 * (Income - 10000);
    else
            TaxSubTotal = 1400 + .04 * (Income - 50000);
    Exemption= nDependents * 50;
    TaxTotal=TaxSubTotal - Exemption;
    // last if - check negative tax
    if (TaxTotal<0) //In case of negative tax
    TaxTotal=0;
    cout << „Result of Tax \n";
    cout << "Tax Bill \n";
    cout << " Citizen's Income: " << Income <<'\n';
    cout << " Tax Subtotal: " << TaxSubTotal << '\n';
    cout << "Number of Dependents: " << nDependents << '\n';
    cout << " Tax Exepmtion: " << Exemption << '\n';
    cout << " Final Tax Bill: " << TaxTotal << '\n';
}
```

First if	**Second if**	**If-else-if**	**Last if**	**Result**
Income < 0	*doesn't matter*	*doesn't matter*	*doesn't matter*	negative income error
Income >= 0	NDependents <= 0	*doesn't matter*	*doesn't matter*	invalid dependents error
Income >= 0	NDependents > 0	Income < 10000	TaxTotal < 0	bracket 1 negative tax
Income >= 0	NDependents > 0	10000 <= Income < 50000	TaxTotal < 0	bracket 2 negative tax
Income >= 0	NDependents > 0	Income >= 50000	TaxTotal < 0	bracket 3 negative tax
Income >= 0	NDependents > 0	Income < 10000	TaxTotal >= 0	bracket 1
Income >= 0	NDependents > 0	10000 <= Income < 50000	TaxTotal >= 0	bracket 2
Income >= 0	NDependents > 0	Income >= 50000	TaxTotal >= 0	bracket 3

Test Cases for this example

Income	NDependents	Expected Result
-5	Doesn't matter	negative income error
0	0	invalid dependents error
100	1	0 (bracket 1, negative tax)
20000	11	0 (bracket 2, negative tax)
50000	100	0 (bracket 3, negative tax)
9000	1	130 (bracket 1)
15000	1	300 (bracket 2)
100000	1	3350 (bracket 3)

Coverage analysis tools:

- Bullseye Coverage
- Cantata++
- CodeTEST
- LOGISCOPE
- Panorama C/C++
- Rational PureCoverage
- TCAT C/C++
- GCT

Reference: http://testingfaqs.org/t-eval.html

Performance profiler

- Code profiling is the process of benchmarking the execution to understand where the time is being spent in terms of code execution
- Which lines of code are responsible for the bulk of execution time?
- How many times is this looping construct executed?
- Which approach to coding a block of logic is more efficient?
  - Without profiling, the answer to the above questions becomes a guessing game.

Facts of profiler

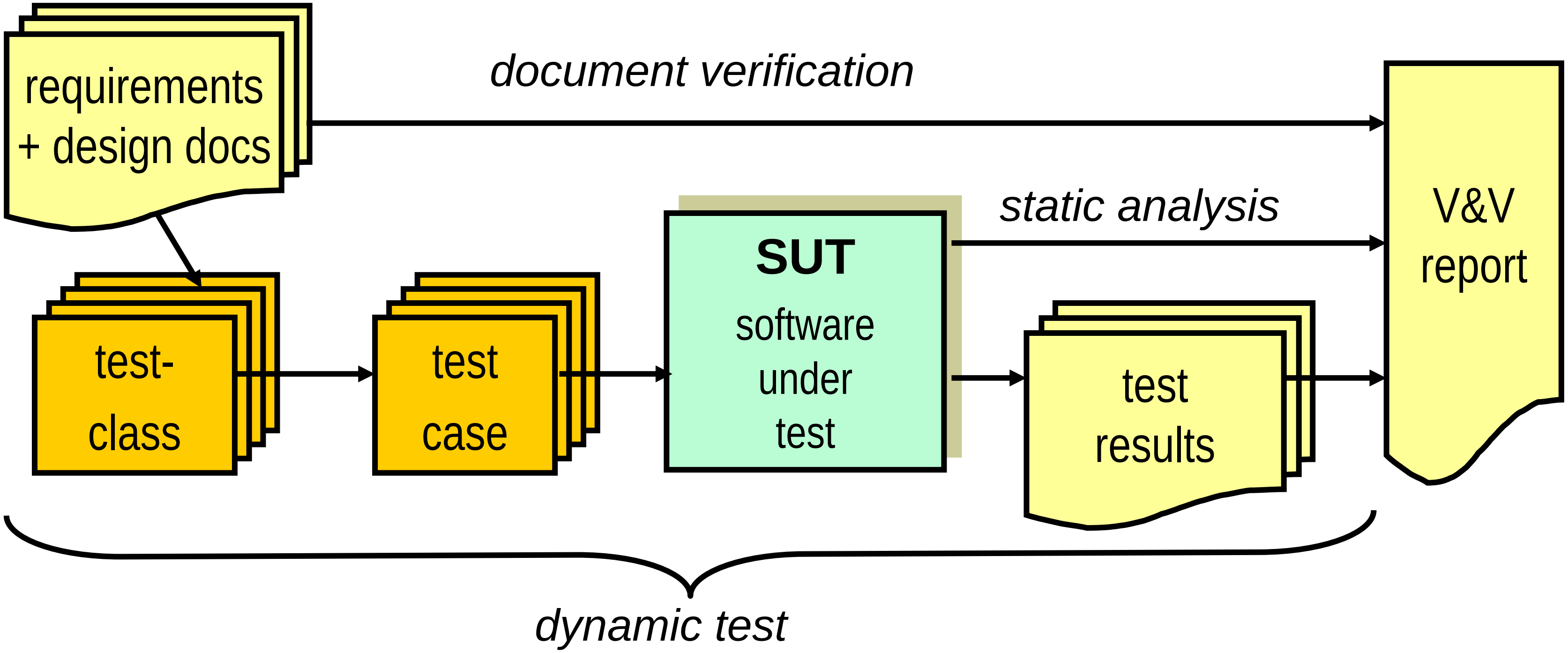
- Why/When we need ?

  Profiler will pinpoint slow code and allow us to drill down to examine it, so code can be optimized and performance can be improved

- How it works ?

  Profiler runs while we are using our application and records the frequency and time spent in each line of code

What is dynamic test



C /C++ testing tool

Professional Software Testing Tools Cantata++

(Tools brief presentation)

Overview Cantata++

- Dynamic Testing
  - Executing the software
  - Under known conditions
  - Verifying results against expected outcomes
- Code Coverage
- Integration Testing
- Host-Target Testing
- Static Analysis metrics
- Generation of code metrics
  - Does source code meets quality and maintainability standards?
- Cantata++ for C and Cantata++ for C++ are two very common industry accepted tools for testing

Unit testing features

- Wizard driven template test script generation
- Automated checking of expected results
- Black / White box techniques
- Simulation of external software (Stubs)
- State transition testing
- Real-time performance analysis
- Automated regression testing

Integration testing additional features

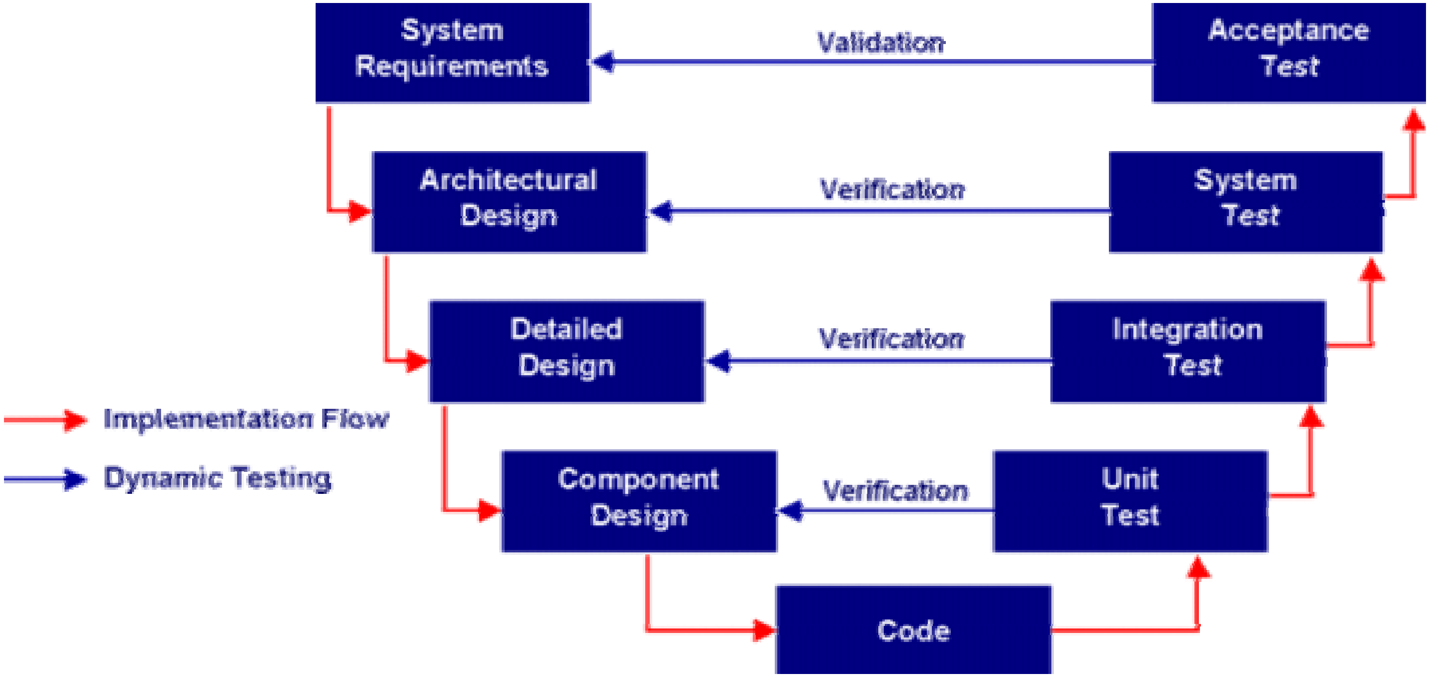
- Total control over external interfaces (Wrapping)
- Call sequence validation
- User observation tests
- Mulit-threaded test execution

Overview - Aims

- Repeatable
- Automatic
- Auditable
- Portable
- Measurable
- Productive and Efficient

Overview – Software Lifecycle

- Software development follows a life-cycle model. The V-Model is a useful example, but there are many different life-cycles.



Cantata++ can be used at all stages of the lifecycle

How stubbing works

Cantata++ Creates a Stub function containing programmable instances for the external object, which are called by the source code under test and replace the original external object (software, firmware or hardware)

Overview – Dynamic Testing

- Host and Target testing
  - Test on development platform and in the target environment
- Regression testing
  - Automated re-execution of tests
- White-box and Black-box testing
  - Testing with and without knowledge of the code
- Isolation testing
  - Testing with simulation of external code interfaces

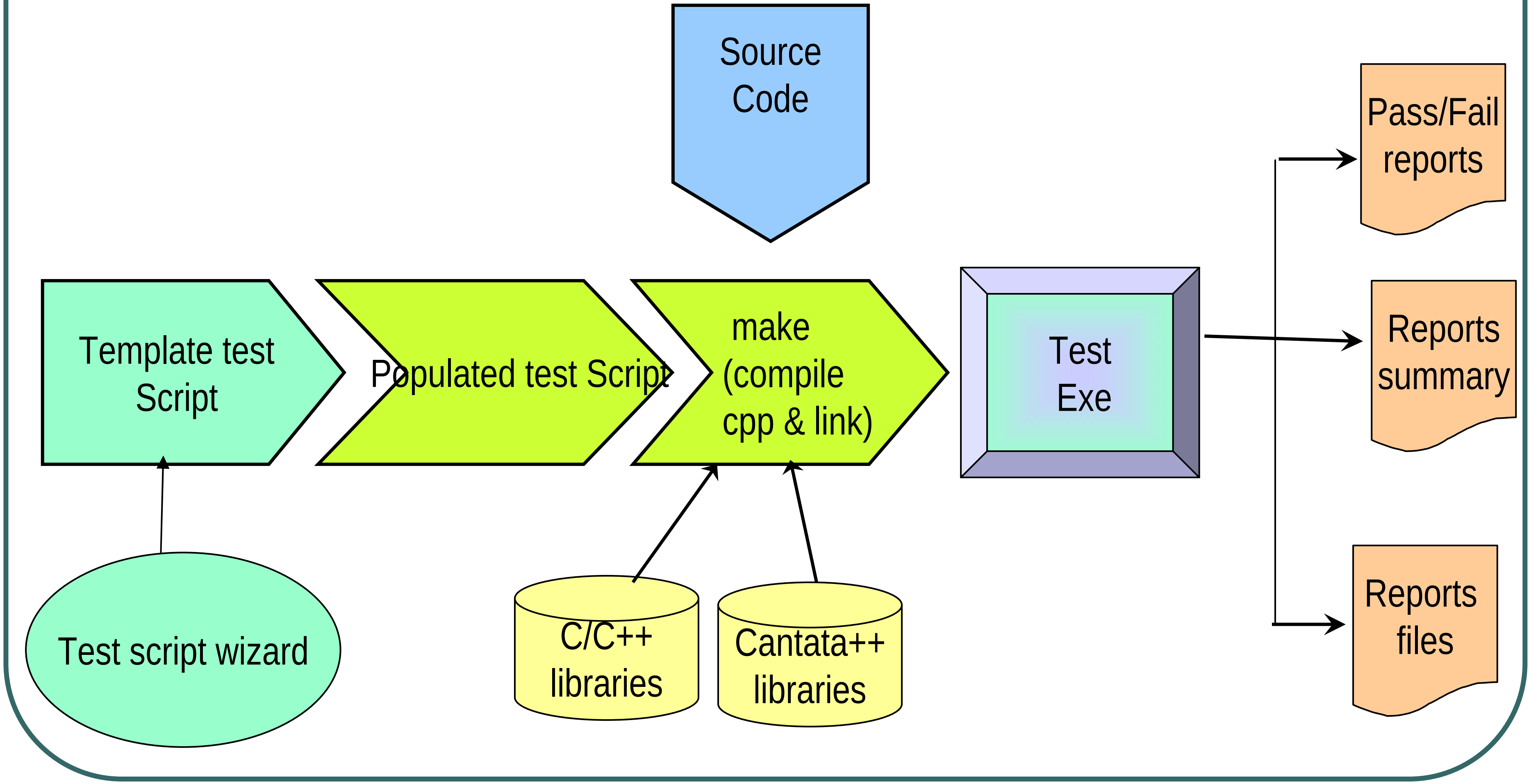
Overview – Coverage Analysis

- Measure and report coverage
- Set a Pass/Fail coverage checks for your project
- Metrics supported include:
  - Entry-Points
  - Statements
  - Decisions
  - Conditions

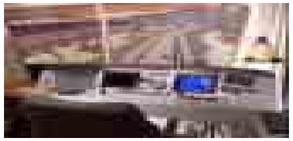
Overview – Static Analysis

- Derives metrics from the source code to help improve its quality
- Output data in CSV format for
  - Numerical or graphical analysis
- Provide objective data for
  - Code reviews, project management, end of project statistics
- Provide facilities to generate up to 300 source code metrics
  - Code construct counts, Complexity metrics, File metrics

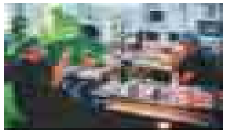
How cantata++ works



Cantata++ Customers



Train Control - Alcatel



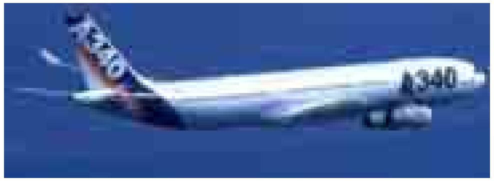
Medical Systems – GE Medical

Cantata++ running under Symbian – Nokia Series 60

Nuclear Reactor Control - Thales

EFA Typhoon – BAe Systems

International Space Station – Dutch Space



Airbus A340 – Ultra Electronics

Further Information

- http://www.iplbath.com

Tool Summary

- **Objective test strategy should achieve**

“an acceptable level of confidence at an acceptable level of cost”.

- **Tests should be**
  - Planned for in the project
  - Against specified requirements
  - As automated and repeatable as possible
  - Maintainable

Summary of testing

- Verify operation at normal parameter values
  (a black box test based on the unit’s requirements)
- Verify operation at limit parameter values
  (black box)
- Verify operation outside parameter values
  (black box)
- Ensure that all instructions execute
  (statement coverage)
- Check all paths, including both sides of all branches
  (decision coverage)

Summary of testing

- Check the use of all called objects
- Verify the handling of all data structures
- Verify the handling of all files
- Check normal termination of all loops

  (part of a correctness proof)
- Check abnormal termination of all loops

Summary of testing

- Check normal termination of all recursions
- Check abnormal termination of all recursions
- Verify the handling of all error conditions
- Check timing and synchronization
- Verify all hardware dependencies

Statements of **(Watts Humphrey)**

The End

Thank You