Automatic Module Testing Concept

MANAGEMENT SUMMARY

White paper that explains some aspects of automatic module testing

TABLE OF CONTENTS

1 Purpose

This document provides some do’s and don’ts on how to automatic (module) testing.

2 Module Testing Introduction

Testing at the end of a development project is always under pressure as the product is always late and need to be shipped, sold, reproduced, distributed, etc.

So you better built in quality so you don’t have to rely on ‘final’ testing as product quality boost.
There are many methods you can use to build quality in your product and one of them is doing module tests.

With module tests I mean automatic testing where the module/component/code is testes in isolation with minimal interaction with other program in automated mode without human interaction.

3 What is automatic module testing

3.1 *The concept of automatic module testing*

With module testing you test the interfaces of the module and its functional behavior. Module testing is mostly done by the developer of the module as he knows what he has built and where the weak spots are.
As the developer knows the internals it is easy to define all kind of special test cases that can search for the limits of the program code.
This testing with knowledge of the internals is called 'White box' testing.
In 'black box' testing you can only use the in and outputs.

A good module test verifies all requirements, interfaces and their boundaries. But also test all design decisions and problem areas of the module making it a sort artificial knowledge unit.

3.2 *Why automated module testing?*

In most cases it will cost almost no effort to execute a module test once it is defined.
You can even combine a number of module tests to nightly regression tests or do them as part of a release built. The options are too many to name.
This makes the 'Return On Investment' for module testing very high.

But probably the biggest benefit is in aftercare.
When the product is in maintenance and the development knowledge has gone, and specifications, designs, etc. are incomplete, outdated or obsolete then these module tests serve as a knowledge container where every code concept violation (during updates) is probably detected.

3.3 *The extra effort to create a module test*

The module is tested in isolation and not in its 'natural' work environment.
This implies that all resources that the module uses need to be simulated to that it can function as intended.

This is easier said then done and not always possible.
E.g. how to simulate hardware, an operating system; interrupt handlers, other modules, etc.?
Generally you make the best of it by creating so called 'stubs'. Stubs are a replacement of the original functionality that mimics external behavior.
A stub will create output for a limited set of specified input conditions. How good and how much depends on the effort you want to put into it.

But stubs will always pose a severe restriction in the things you want to test on a module.

Reuse of stubs in the same project can solve some portion of this problem.

3.4 How to encapsulate the target module

Intrusive test code:
I assume that you can use the module code for testing as it would be used for the original program. That implies that no additional test code needs to be inserted in the original source.

If you really need some intrusive test code in the source consider using defines in favor of compiler switches. But you want to make sure that the additional test code has no influence on the normal code execution!

Component file structure:
The following file structure is mostly used

```
<Module-Name>
        +------------ source-code
        +------------ documentation
        +------------ third-party-code  (maybe it has third party specific sub folders)
        +------------ test-code
        +------------ test-reports
```

Test code compilation:
The best solution is to include the target module source code via an include statement.

4 General Statements

- The executed test scripts run without human interaction (After start-up) Scripts can be C, C++, Perl, MS Batch, Bourne shell code or another (scripting) language!
- Test scripts are put under version control
- The CM tool version number is the version of the test script. (Makes it easier to maintain.)

5 Test Script Specification

- Specific availability of resources should be tested at the start of the script execution. Preferably this is also script runtime option.
- Every test script has an exit code indicating the test result of the complete test

    - Exit code 0
      The ALL EXECUTED test cases succeeded.
      (For each test case the target is met.)
    - Exit code >0
      At least one test case has failed.
      (A higher number might be an indicator for a test case or test cluster, e.g. 1 for cluster1, 2 for cluster2, etc.)
- Test Specifications
    - The test specification is imbedded in the test script
      Can be easily recognized when the test script is opened.
    - The test report is also a test specification
      Run a dummy test to get the (essence) test specification
      Optional an test option can be passed to generate a symbolic output including all test case specifications
    - In the script some more elaborate descriptions (in comment) might be needed

6 Runtime options to pass to a test script

Test script runtime options.
(How to pass this option is based on the test implementation!)

- <None>
  Execute the test script in default mode (Preferably all tests)
- Dummy
  Do not test. Display all defined test cases so that the output can be used as a test specification.
- Confidence
  Execute a limited/predefined set of all test cases to show global confidence that the 'unit' is still working properly. (Regression test)
  Can be used if the total test time will take to long!
- Resources
  Check the availability of resources that are needed to execute some specific (or all) tests. E.g. Power supply, antenna signal, input streams, etc.

7 Automatic Test Report Generation

- In the test report header is described
    - The name of the module/unit/layer/etc. is present
    - The name of the test is present
    - The purpose of the test is described (if not clear from the name)
    - The version of the test specification used is present
    - The version of the product/component is present
    - Start Date & Time of the test
- Each test case should be on a single line specifying

- Test number
- Test name /description
- Valid range (if specified and other then binary Yes/No.)
- Value (other then binary. For binaries Yes/No|On/Off)
- Test result
    - i. Succeeded (Passed is a reserved word for the end result)
    - ii. Failed (Flunked)
    - iii. Not executed

- Every run must list all available test cases
  If a case is 'Not executed' it still must be present in the report.
- The last line of the report indicates the complete test result: Passed or Failed.
  Passed is only valid when all tests were successful
- No real footer used to support a 'grep' on the word Failed. (Last line report!)
  (Failed is a reserved word and can't be used in test descriptions / specification!)

The test output should be in plain ASCI, 'simple' html or CSV format.

The words: Passed, Succeeded, Failed, Not Executed are reserved words and should not be used in the report for any other reason as defining test results.

*

ImQuSo Improved Quality Solutions.
Po Box 169
5540 AD REUSEL, The Netherlands

Doc. ImQuSo-WP-2008-010 Rev. 0.3 2008-05-05