OOWG Tutorial
Software testing for the VLT

G.Chiozzi

16/07/96

Contents

This is a generic introduction to software testing for the VLT and does not cover only OO strictly related matters.

Introduction

Software testing has the fundamental purpose
of proofing and certifying
that a piece of software
behaves according to the specifications.

This requires that the testing phase is fully integrated in the development process and that formal methods are available to trace specifications in the tests.

This is a very difficult task, since typically the specification documents are not detailed and clear enough to provide a good reference frame for testing.
Pragmatic countermeasures must then be taken to make up for the lack in specifications.

Testing different levels of software, from basic libraries to complete applications, presents different problems and requirements.

Object-oriented software again introduces a complete new set of problems and requires a different and new approach to testing.

Software levels

It is possible to group software components in levels based on their complexity and on the type of usage. Any of these levels present different testing peculiarities.

In an Object-Oriented development environment we can identify the following levels:

- Basic function library
  A set of "classical" independent functions
- Class library
  A set of classes designed using OO techniques (or groups of functions related by a class-like structure).
- Design pattern
  A set of cooperating classes with a specification of their interaction rules to solve a specific problem.
- Application framework
  A skeleton for a typical application, based on design patterns
- Final application
  The final application software for a user

Function library

A function library is from many respects the easier to test:

- A single function is typically independent from the other functions
- Functions typically do not depend on time: the input uniquely specify the expected output.
- Specifications for libraries are usually clear and well done, since they come from software developers.

Testing a function library typically requires:

- Testing the functions one by one
- Providing to functions (exhaustive) sets of input values
- Verifying pre- and post-conditions
- Checking that the generated output is the specified one.

Class library

In the Object-Oriented way of thinking, data and behaviour are grouped in a single entity (the Class), responsible for providing a set of services to an external user.

This applies not only to code developed with OO languages, but also to code developed with standard languages but designed with these ideas in mind.

This rises a full new set of problems:

- An object has a life over time and the result of a call to a method typically depends on the history and internal state of the object.
- Each lower level in an inheritance hierarchy is a new context for inherited features.
- Polymorphism with dynamic binding dramatically increase the number of possible execution paths.
- The shift to composition adds another dimension to OO software testing.

Static analysis of source code to identify paths is of relatively little help here

Testing a class library requires:

- Testing the classes one by one
- Testing all the methods in a class, as if they functions
- Testing "sequences" of method's calls to verify the behaviour of classes with time and the dependencies between subsequent method's calls

It would be also very important to provide "test tools" to test classes when they are used in a different context or through inheritance. Each class should have a companion test class or a Test() method used for this purpose.

Design pattern and Application framework

These structures introduce a much higher level of complexity, since they involve the interaction between many classes and model specific applicative problems.

- These systems are inherently dynamic and event driven.
- Components offered for reuse should be highly reliable and thus require extensive testing.
- Each reuse is a new context and re-testing is prudent

On the other end:

- Widely reused components benefits from indirect testing inside applications.
- Problems are moved from final application level to a lower level, more controllable.

The testing required by a single component is higher than in a traditional system, but the number of components in a final application is much smaller.

Testing a design pattern or an application framework typically requires:

- Testing the new functions and classes in the usual way, one by one.
- Providing “black box” testing.
- Providing behavioural testing of sample applications using the pattern/framework.

Final application

Applications are for final users

This makes testing much more difficult, since:

- Specifications are generic and unclear, so they can be interpreted in many ways.
- Complex system can easily produce unanticipated results
- In a distributed real-time system, correctness depends also on the timing constraints imposed by the real-world processes

To this, we have to add the specific set of testability problems imposed by user interfaces.

Testing an application requires:

- Testing all new functions, classes, patterns in the usual way, one by one
- Providing behavioural tests that exercise the external interface of the application.

VLT Software Modules

**The basic elementary unit
for VLT software components
is a
MODULE**

A module can contain:

- a function library `(msg, db, evt)`
- a class library or an application framework `(eccs, evh)`
- an application `(prs, msw)`
- sets of configuration data for an installation `(tcsint)`

Every module must be tested as an independent unit, providing a modular test.

The standard module's directory structure includes a `/test` directory for this purpose.

Modules are integrated in higher level units `(CCS,TCS)`.

An integration test must be provided.

Automatic regression testing

Our aim is to have
fully automatic regression testing
both for modules and integration

This makes easier and safer the validation of new versions of the software, but it is not always possible, in particular for interactive applications.

Every modular/integration test should include a:

- Modular test plan
- Modular test report

The Modular test plan includes a detailed description of the test scope, setup and procedures. It is essential when the test requires manual operation.

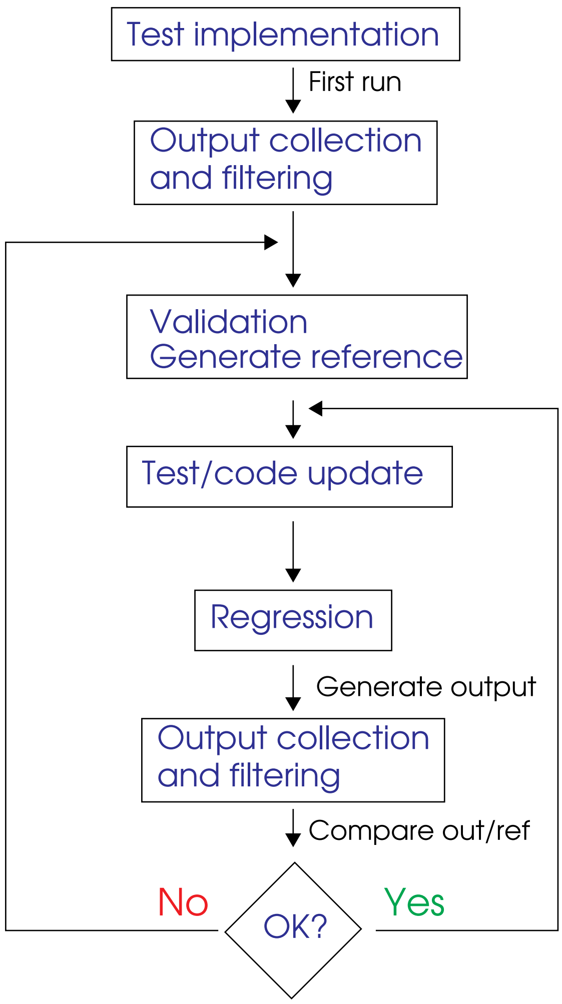
It should also include a traceability matrix
with the functional specifications.

The Modular test report should be issued and updated after major testing sessions (like a new release) and contain indication of all problems encountered during testing.

The test should not be designed and implemented by the module's responsible and any person with VLT software knowledge should be able to run it.

Output-based testing

Testing procedures are not self-contained
Test is based on first-run user validation of the output



This approach:

- Makes easier writing test programs
- Makes easier integrating real-world examples in the test

but:

- Requires accurate checking of ref/diff files

Tools are provided to support test's implementation

tat:

- environment management
- basic output filtering capabilities
- software build/cleanup
- test selection

eccsTestDriver:

- multi process tests
- advanced filtering capabilities
- test selection

Some effort is necessary to better integrate the two tools

Example:

This example shows how tat and eccsTestdriver are used together in the test for the TCS Mode Switching Module:

- TestList (tat init file)

```
ENVIRONMENT wsTat WS
mswTat
```

- mswTat

```
mswTestInitDb.sh
TestDriver
```

- TESTLIST (eccsTestDriver init file)

```
1 mswTestStdCommands mswControl  "evhDummyPRS mswTestDummyPRS.dat" "evhDummyTRK mswTestDummyTRK.dat" "evhDummyACT mswTestDummyACT.dat" "evhDummyLT5M3 mswTestDummyLT5M3.dat" mswTestStdCommands
2 mswTestStateChanges mswControl "evhDummyPRS mswTestDummyPRS.dat" "evhDummyTRK mswTestDummyTRK.dat" "evhDummyACT mswTestDummyACT.dat" "evhDummyLT5M3 mswTestDummyLT5M3.dat" mswTestStateChanges
3 mswTestFlagsRequiredIgnore mswControl "evhDummyPRS mswTestDummyPRS.dat" "evhDummyTRK mswTestDummyTRK.dat" "evhDummyACT mswTestDummyACT.dat" "evhDummyLT5M3 mswTestDummyLT5M3.dat" mswTestFlagsRequiredIgnore
```

- TestDriver.sed (sed filter for eccsTestDriver)

```
s/Environment  : .* StackId  : [ 0-9][ 0-9][ 0-9][ 0-9][ 0-9][ 0-9]/Environment  : -te----  StackId  : ------/g
s/Sequence : *[ 0-9][ 0-9]/Sequence : --  /g
s/signal: [ 0-9][ 0-9]/signal: --/g
s/[lw]t[e0-9][a-z0-9A-Z]\{2,4\}/-te----/g
s/@(#) $Id.*/@(#) $Id/g
```

Application's testing

The handling of multi-process tests provided by eccsTestDriver is extremely important while testing applications.

- Applications provide a command interface.
- There can be many processes interacting within the application itself or as part of other applications.
- In order to provide independent modular testing, processes contained in other modules must be simulated and/or replaced by dummies.

A typical "library test" consists of programs written in the same language of the library (typically C/C++ or TCL) that directly exercise the provided function and classes.

An "application test" must execute the application and exercise its external interface, sending commands and using dummies for the other applications it is supposed to talk to.

Example:

- Entry in TESTLIST (eccsTestDriver init file)

```
1 mswTestStdCommands mswControl  "evhDummyPRS mswTestDummyPRS.dat" "evhDum-
myTRK mswTestDummyTRK.dat" "evhDummyACT mswTestDummyACT.dat" "evhDummyLT5M3
mswTestDummyLT5M3.dat" mswTestStdCommands
```

- mswTestStdCommand (ctrl sequencer script)

```
#------------------------------------------------------------------------
printLog "Test standard commands"
mswTestInit

#------------------------------------------------------------------------
# Test standard commands one by one
printLog "Test standard commands one by one"

mswTestSendCommandAndGetStatus PING
mswTestSendCommandAndGetStatus INIT

mswTestSendCommandAndGetStatus VERBOSE On
testReadDbValue ":Appl_data:TCS:msw.verbose"

mswTestSendCommandAndGetStatus VERBOSE Off
testReadDbValue ":Appl_data:TCS:msw.verbose"

mswTestSendCommandAndGetStatus STOPSIM
testReadDbValue ":Appl_data:TCS:msw.simulation"

testReadModuleState
mswTestSendCommandAndGetStatus OFF
mswTestSendCommandAndGetStatus ONLINE
testReadModuleState

mswTestExit
# ____oOo____
```

- mswTestDummy.dat (dummy config file; some tests use also dynamic dummy configuration)

```
# General commands (for standard modules)
SETUP,"OK",0,0,1

# Mode Switching emulation
PRESET,"OK",0,0,10
EPRESET,"OK",0,0,10

# Tracking emulation
STOP,"OK",0,0,5

OBJSTAR,"OK",0,0,5

OFFSAD,"OK",0,0,5
OFFSADG,"OK",0,0,5
......
```

- mswTestStdCommands.ref (test's ref. file)

```
2 - Entering the main loop and waiting for commands!
4 - Loading COMMANDS file mswTestDummyACT.dat
2 - Loading COMMANDS file mswTestDummyPRS.dat
4 - Loaded 5 commands
2 - Loaded 5 commands
5 - Entering the main loop and waiting for commands!
5 - Loading COMMANDS file mswTestDummyLT5M3.dat
3 - Entering the main loop and waiting for commands!
5 - Loaded 6 commands
3 - Loading COMMANDS file mswTestDummyTRK.dat
3 - Loaded 5 commands
1 - mswControl - Application started (proc name: mswControl, DB root point:
:Appl_data:TCS)
6 - Test standard commands
6 - Starting test
..........
```

Other support tools

Other tools are provided as part of the VLT software or as external tools to help in testing, like for example:

- **User interface testing library**
  To implement automatic testing of UIF
- **evhDummyControl**
  To interactively configure dummy programs
- **PURIFY**
  To perform dynamical memory tests
- **PURECOV**
  To measure test coverage

Reference documents and readings

- **VLT Software Management Plan**
  VLT-PLA-ESO-00000-0006
- **VLT Software tools for Automated Testing**
  VLT-MAN-ESO-17200-0908
- **Extended CCS User Manual**
  VLT-MAN-ESO-17210-0770
- **TCS Mode Switching Modular Test Plan**
  VLT-VER-ESO-17230-0895
- **TCS Mode Switching Test Report**
  VLT-VER-ESO-17230-0896
- **Object-oriented software testing**
  Special issue of Communication of the ACM, September 1994
- **Debugging for timing constraints violation**
  J.P.Tsai Y.Bi S.Y.Yang, IEEE Software, March '96
- **Performance testing C++ code**
  N.Hunt, JOOP, January '96
- **Automatically tracking test case execution**
  N.Hunt, JOOP, Nov.-Dec. '95

Conclusion

In the VLT software a lot of effort is spent in providing a good testing coverage in all levels of software development, from basic libraries to application software.

The methodology used includes:

- Modular testing
- Behavioural testing
- Integration testing

and makes use as much as possible of:

- Automatic regression testing
- Test analysis and coverage tools
- Formal checks against functional specification and design documentation

More effort should be spent in providing built-in test capability in reusable software and in particular inside object oriented software (classes, design patterns...).