

Organisational Approaches for Unit Testing

Executive Summary

This paper describes three organisational approaches for unit testing: **top down**, **bottom up** and **isolation**. The organisational approach is a key element of unit test strategy and planning; selection of an inappropriate approach can have a significant impact on the cost of unit testing and software maintenance. A unit test strategy based on isolation testing is recommended.

IPL is an independent software house founded in 1979 and based in Bath. IPL was accredited to ISO9001 in 1988, and gained TickIT accreditation in 1991. Both Cantata and AdaTEST have been produced to these standards.

Copyright

This document is the copyright of IPL Information Processing Ltd. It may not be copied or distributed in any form, in whole or in part, without the prior written consent of IPL.

*IPL
Eveleigh House
Grove Street
Bath
BA1 5LR
UK*

*Phone: +44 (0) 1225 444888
Fax: +44 (0) 1225 444400
email ipl@iplbath.com*



Certificate Number FM 01589

*Last Update: 03/07/1997 08:29:00
File: ORG_UT.DOC*

1. Introduction

Unit testing is the testing of individual components (units) of the software. Unit testing is usually conducted as part of a combined code and unit test phase of the software lifecycle, although it is not uncommon for coding and unit testing to be conducted as two distinct phases.

The basic units of design and code in Ada, C and C++ programs are individual subprograms (procedures, functions, member functions). Ada and C++ provide capabilities for grouping basic units together into packages (Ada) and classes (C++). Unit testing for Ada and C++ usually tests units in the context of the containing package or class.

When developing a strategy for unit testing, there are three basic organisational approaches that can be taken. These are **top down**, **bottom up** and **isolation**. These three approaches are described and their advantages and disadvantages discussed in sections 2, 3, and 4 of this paper.

The concepts of **test drivers** and **stubs** are used throughout this paper. A **test driver** is software which executes software in order to test it, providing a framework for setting input parameters, executing the unit, and reading the output parameters. A **stub** is an imitation of a unit, used in place of the real unit to facilitate testing.

An AdaTEST or Cantata test script comprises a test driver and an (optional) collection of stubs. Using AdaTEST or Cantata to implement the organisational approaches to unit testing presented in this paper is discussed in section 5.

2. Top Down Testing

2.1. Description

In **top down** unit testing, individual units are tested by using them from the units which call them, but in isolation from the units called.

The unit at the top of a hierarchy is tested first, with all called units replaced by stubs. Testing continues by replacing the stubs with the actual called units, with lower level units being stubbed. This process is repeated until the lowest level units have been tested. Top down testing requires test stubs, but not test drivers.

Figure 2.1 illustrates the test stubs and tested units needed to test unit D, assuming that units A, B and C have already been tested in a top down approach.

A unit test plan for the program shown in figure 2.1, using a strategy based on the top down organisational approach, could read as follows:

Step (1)

Test unit A, using stubs for units B, C and D.

Step (2)

Test unit B, by calling it from tested unit A, using stubs for units C and D.

Step (3)

Test unit C, by calling it from tested unit A, using tested units B and a stub for unit D.

Step (4)

Test unit D, by calling it from tested unit A, using tested unit B and C, and stubs for units E, F and G. (Shown in figure 2.1).

Step (5)

Test unit E, by calling it from tested unit D, which is called from tested unit A, using tested units B and C, and stubs for units F, G, H, I and J.

Step (6)

Test unit F, by calling it from tested unit D, which is called from tested unit A, using tested units B, C and E, and stubs for units G, H, I and J.

Step (7)

Test unit G, by calling it from tested unit D, which is called from tested unit A, using tested units B, C, E and F, and stubs for units H, I and J.

Step (8)

Test unit H, by calling it from tested unit E, which is called from tested unit D, which is called from tested unit A, using tested units B, C, E, F and G, and stubs for units I and J.

Step (9)

Test unit I, by calling it from tested unit E, which is called from tested unit D, which is called from tested unit A, using tested units B, C, E, F, G and H, and a stub for units J.

Step (10)

Test unit J, by calling it from tested unit E, which is called from tested unit D, which is called from tested unit A, using tested units B, C, E, F, G, H and I.

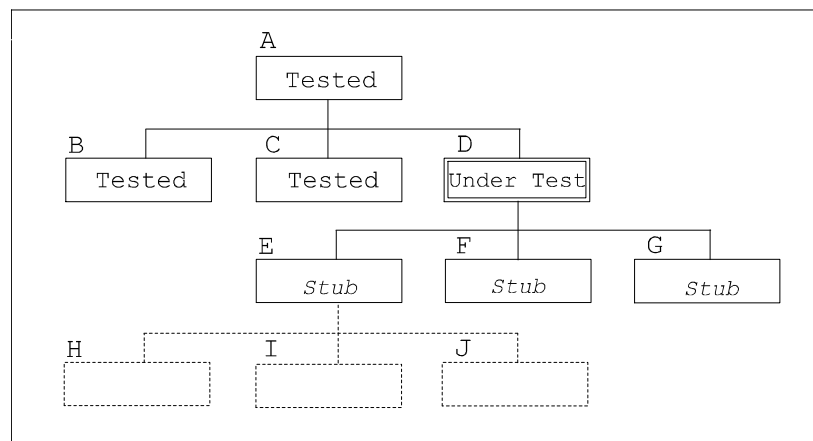


Figure 2.1 - Top Down Testing

2.2. Advantages

Top down unit testing provides an early integration of units before the software integration phase. In fact, top down unit testing is really a combined unit test and software integration strategy.

The detailed design of units is top down, and top down unit testing implements tests in the sequence units are designed, so development time can be shortened by overlapping unit testing with the detailed design and code phases of the software lifecycle.

In a conventionally structured design, where units at the top of the hierarchy provide high level functions, with units at the bottom of the hierarchy implementing details, top down unit testing will provide an early integration of 'visible' functionality. This gives a very requirements oriented approach to unit testing.

Redundant functionality in lower level units will be identified by top down unit testing, because there will be no route to test it. (However, there can be some difficulty in distinguishing between redundant functionality and untested functionality).

2.3. Disadvantages

Top down unit testing is controlled by stubs, with test cases often spread across many stubs. With each unit tested, testing becomes more complicated, and consequently more expensive to develop and maintain.

As testing progresses down the unit hierarchy, it also becomes more difficult to achieve the good structural coverage which is essential for high integrity and safety critical applications, and which are required by many standards. Difficulty in achieving structural coverage can also lead to a confusion between genuinely redundant functionality and untested functionality. Testing some low level functionality, especially error handling code, can be totally impractical.

Changes to a unit often impact the testing of sibling units and units below it in the hierarchy. For example, consider a change to unit D. Obviously, the unit test for unit D would have to change and be repeated. In addition, unit tests for units E, F, G, H, I and J, which use the tested unit D, would also have to be repeated. These tests may also have to change themselves, as a consequence of the change to unit D, even though units E, F, G, H, I and J had not actually changed. This leads to a high cost associated with retesting when changes are made, and a high maintenance and overall lifecycle cost.

The design of test cases for top down unit testing requires structural knowledge of when the unit under test calls other units. The sequence in which units can be tested is constrained by the hierarchy of units, with lower units having to wait for higher units to be tested, forcing a 'long and thin' unit test phase. (However, this can overlap substantially with the detailed design and code phases of the software lifecycle).

The relationships between units in the example program in figure 2.1 is much simpler than would be encountered in a real program, where units could be referenced from more than one other unit in the hierarchy. All of the disadvantages of a top down approach to unit testing are compounded by a unit being referenced from more than one other unit.

2.4. Overall

A top down strategy will cost more than an isolation based strategy, due to complexity of testing units below the top of the unit hierarchy, and the high impact of changes. The top down organisational approach is not a good choice for unit testing. However, a top down approach to the integration of units, where the units have already been tested in isolation, can be viable.

3. Bottom up Testing

3.1. Description

In **bottom up** unit testing, units are tested in isolation from the units which call them, but using the actual units called as part of the test.

The lowest level units are tested first, then used to facilitate the testing of higher level units. Other units are then tested, using previously tested called units. The process is repeated until the unit at the top of the hierarchy has been tested. Bottom up testing requires test drivers, but does not require test stubs.

Figure 3.1 illustrates the test driver and tested units needed to test unit D, assuming that units E, F, G, H, I and J have already been tested in a bottom up approach.

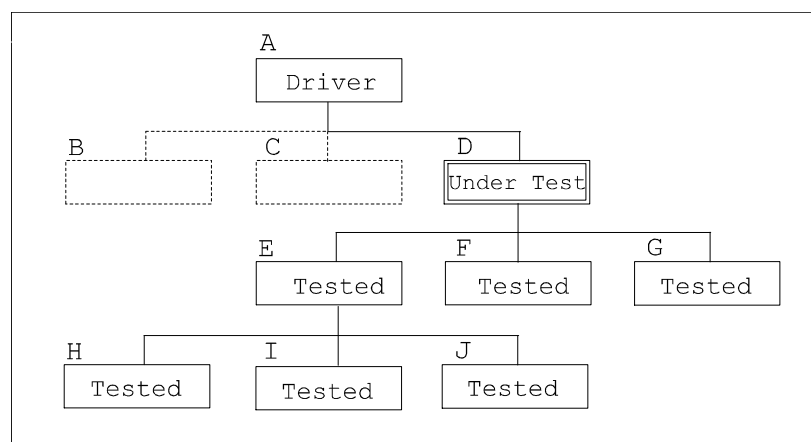


Figure 3.1 - Bottom Up Testing

A unit test plan for the program shown in figure 3.1, using a strategy based on the bottom up organisational approach, could read as follows:

Step (1)

(Note that the sequence of tests within this step is unimportant, all tests within step 1 could be executed in parallel.)

Test unit H, using a driver to call it in place of unit E;

Test unit I, using a driver to call it in place of unit E;

Test unit J, using a driver to call it in place of unit E;

Test unit F, using a driver to call it in place of unit D;

Test unit G, using a driver to call it in place of unit D;

Test unit B, using a driver to call it in place of unit A;

Test unit C, using a driver to call it in place of unit A.

Step (2)

Test unit E, using a driver to call it in place of unit D and tested units H, I and J.

Step (3)

Test unit D, using a driver to call it in place of unit A and tested units E, F, G, H, I and J. (Shown in figure 3.1).

Step (4)

Test unit A, using tested units B, C, D, E, F, G, H, I and J.

3.2. Advantages

Like top down unit testing, bottom up unit testing provides an early integration of units before the software integration phase. Bottom up unit testing is also really a combined unit test and software integration strategy. All test cases are controlled solely by the test driver, with no stubs required. This can make unit tests near the bottom of the unit hierarchy relatively simple. (However, higher level unit tests can be very complicated).

Test cases for bottom up testing may be designed solely from functional design information, requiring no structural design information (although structural design information may be useful in achieving full coverage). This makes the bottom up approach to unit testing useful when the detailed design documentation lacks structural detail.

Bottom up unit testing provides an early integration of low level functionality, with higher level functionality being added in layers as unit testing progresses up the unit hierarchy. This makes bottom up unit testing readily compatible with the testing of objects.

3.3. Disadvantages

As testing progresses up the unit hierarchy, bottom up unit testing becomes more complicated, and consequently more expensive to develop and maintain. As testing progresses up the unit hierarchy, it also becomes more difficult to achieve good structural coverage.

Changes to a unit often impact the testing of units above it in the hierarchy. For example, consider a change to unit H. Obviously, the unit test for unit H would have to change and be repeated. In addition, unit tests for units A, D and E, which use the tested unit H, would also have to be repeated. These tests may also have to change themselves, as a consequence of the change to unit H, even though units A, D and E had not actually changed. This leads to a high cost associated with retesting when changes are made, and a high maintenance and overall lifecycle cost.

The sequence in which units can be tested is constrained by the hierarchy of units, with higher units having to wait for lower units to be tested, forcing a 'long and thin' unit test phase. The first units to be tested are the last units to be designed, so unit testing cannot overlap with the detailed design phase of the software lifecycle.

The relationships between units in the example program in figure 2.2 is much simpler than would be encountered in a real program, where units could be referenced from more than one other unit in the hierarchy. As for top down unit testing, the disadvantages of a bottom up approach to unit testing are compounded by a unit being referenced from more than one other unit.

3.4. Overall

The bottom up organisational approach can be a reasonable choice for unit testing, particularly when objects and reuse are considered. However, the bottom up approach is biased towards functional testing, rather than structural testing. This can present difficulties in achieving the high levels of structural coverage essential for high integrity and safety critical applications, and which are required by many standards.

The bottom up approach to unit testing conflicts with the tight timescales required of many software developments. Overall, a bottom up strategy will cost more than an isolation based strategy, due to complexity of testing units above the bottom level in the unit hierarchy and the high impact of changes.

4. Isolation Testing

4.1. Description

Isolation testing tests each unit in isolation from the units which call it and the units it calls.

Units can be tested in any sequence, because no unit test requires any other unit to have been tested. Each unit test requires a test driver and all called units are replaced by stubs. Figure 4.1 illustrates the test driver and tested stubs needed to test unit D.

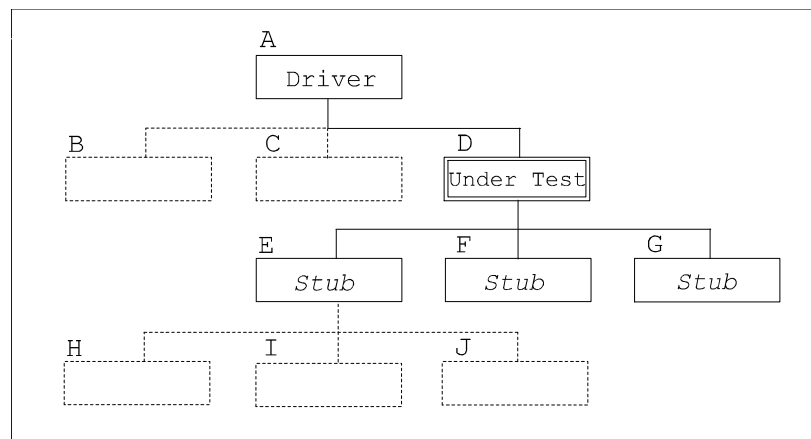


Figure 4.1 - Isolation Testing

A unit test plan for the program shown in figure 4.1, using a strategy based on the isolation organisational approach, need contain only one step, as follows:

Step (1)

(Note that there is only one step to the test plan. The sequence of tests is unimportant, all tests could be executed in parallel.)

Test unit A, using a driver to start the test and stubs in place of units B, C and D;

Test unit B, using a driver to call it in place of unit A;

Test unit C, using a driver to call it in place of unit A;

Test unit D, using a driver to call it in place of unit A and stubs in place of units E, F and G, (Shown in figure 3.1);

Test unit E, using a driver to call it in place of unit D and stubs in place of units H, I and J;

Test unit F, using a driver to call it in place of unit D;

Test unit G, using a driver to call it in place of unit D;

Test unit H, using a driver to call it in place of unit E;

Test unit I, using a driver to call it in place of unit E;

Test unit J, using a driver to call it in place of unit E.

4.2. Advantages

It is easier to test an isolated unit thoroughly, where the unit test is removed from the complexity of other units. Isolation testing is the easiest way to achieve good structural coverage, and the difficulty of achieving good structural coverage does not vary with the position of a unit in the unit hierarchy.

Because only one unit is being tested at a time, the test drivers tend to be simpler than for bottom up testing, while the stubs tend to be simpler than for top down testing.

With an isolation approach to unit testing, there are no dependencies between the unit tests, so the unit test phase can overlap the detailed design and code phases of the software lifecycle. Any number of units can be tested in parallel, to give a 'short and fat' unit test phase. This is a useful way of using an increase in team size to shorten the overall time of a software development.

A further advantage of the removal of interdependency between unit tests, is that changes to a unit only require changes to the unit test for that unit, with no impact on other unit tests. This results in a lower cost than the bottom up or top down organisational approaches, especially when changes are made.

An isolation approach provides a distinct separation of unit testing from integration testing, allowing developers to focus on unit testing during the unit test phase of the software lifecycle, and on integration testing during the integration phase of the software lifecycle. Isolation testing is the only pure approach to unit testing, both top down testing and bottom up testing result in a hybrid of the unit test and integration phases.

Unlike the top down and bottom up approaches, the isolation approach to unit testing is not affected by a unit being referenced from more than one other unit.

4.3. Disadvantages

The main disadvantage of an isolation approach to unit testing is that it does not provide any early integration of units. Integration has to wait for the integration phase of the software lifecycle. (Is this really a disadvantage?).

An isolation approach to unit testing requires structural design information and the use of both stubs and drivers. This can lead to higher costs than bottom up testing for units near the bottom of the unit hierarchy. However, this will be compensated by simplified testing for units higher in the unit hierarchy, together with lower costs each time a unit is changed.

4.4. Overall

An isolation approach to unit testing is the best overall choice. When supplemented with an appropriate integration strategy, it enables shorter development timescales and provides the lowest cost, both during development and for the overall lifecycle.

Following unit testing in isolation, tested units can be integrated in a top down or bottom up sequence, or any convenient groupings and combinations of groupings. However, a bottom up integration is the most compatible strategy with current trends in object oriented and object biased designs.

An isolation approach to unit testing is the best way of achieving the high levels of structural coverage essential for high integrity and safety critical applications, and which are required by many standards. With all the difficult work of achieving good structural coverage achieved by unit testing, integration testing can concentrate on overall functionality and the interactions between units.

5. Using AdaTEST and Cantata

A unit test will be repeated many times throughout the software lifecycle, both during the development part of the lifecycle and later during maintenance. A test harness such as AdaTEST or Cantata can be used to automate unit tests, resulting in unit tests which are easy to repeat and have a low cost of repetition, and reducing risk of human error.

AdaTEST and Cantata test scripts comprise a test driver and an (optional) collection of stubs. AdaTEST and Cantata can be used with any of the organisational approaches to unit testing described by this paper, or with any combination of organisational approaches, enabling the developer to adopt a testing strategy best suited to the needs of a project.

Two related papers are available from IPL:

- Achieving Testability when using Ada Packaging and Data Hiding Methods
- Testing C++ Objects

The paper "Testing C++ Objects" also provides detail about how the complexity of separate class and containment hierarchies leads to problems with a bottom up approach to unit testing. It describes how an isolation approach to unit testing is the only practical way to deal with separate class and containment hierarchies.

6. Conclusion

In practice, it is unlikely that any single approach to unit testing can be used exclusively. Typically, an isolation approach to unit testing is modified with some bottom up testing, in which the called units are a mixture of stubs and already tested real units. For example, it makes more sense for a mathematical function to be used directly, provided that it has already been tested and is unlikely to change.

The recommended strategy is:

- Base your unit test strategy on the isolation approach, then integrate groups of tested units bottom up.
- Compromise by incorporating some bottom up where it is convenient (for example, using real operators, mathematical functions, string manipulation etc.), but remember the potential impact of changes.

This will result in the lowest cost; both to develop unit tests, and to repeat and maintain tests following changes to units, whilst also facilitating the thorough test coverage necessary to achieve reliable software.

Remember that unit testing is about testing units, and that integration testing is about testing the interaction between tested units.