

## Designing Testable Ada

### Executive Summary

It is theoretically possible to test any design. In practice, the cost and efficiency of testing “un-testable” software can be prohibitive. Testing is much easier when the system and the software are designed with testability in mind. This paper provides guidelines for designing testable Ada software.

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. IPL has developed and supplies the AdaTEST and Cantata software verification products. AdaTEST and Cantata 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](mailto:ipl@iplbath.com)*



Certificate Number FM 01589

Last Update:03/07/1997 08:31:00  
File: TSTADA.DOC

## 1. Introduction

It is theoretically possible to test any design. In practice, the cost and efficiency of testing “un-testable” software can be prohibitive.

It is surprising how many software developers do not consider testability of software until it hits them on the head during the unit test phase. At this point those who have designed un-testable software may declare unit testing to be uneconomical and proceed straight on to integration. They may also decide that the way to solve their problem is to use a testing tool, blind to the fact that un-testable is un-testable, irrespective of what tools are applied to the problem.

Tools such as AdaTEST can provide enormous benefit, but only if their use is planned from an early stage in the project and the software designed with testability in mind.

- Requirements should be phrased so as to be testable;
- The architecture should break the system down into testable components;
- The detailed design should specify units which are structured for testability;
- Detailed design and programming guidelines should ensure that units are implemented in a testable way.

Designing for testability provides benefit before the software is tested. Just thinking about testability will improve the structure and legibility of software, reducing the number of bugs even before the actual tests are designed. Once software is complete it has to be maintained, and a significant aspect of maintenance is the maintenance of tests, so designing for testability will also promote maintainability.

This paper provides guidelines for designing testable Ada software. Many of these guidelines represent “good engineering practice”, even if it were not for testability. Although this paper focuses on unit testing, many of the principles are applicable to all levels of testing. Readers should bear in mind that these guidelines require intelligent interpretation for individual circumstances. There are no "black and white" rules about design.

Although IPL supplies the AdaTEST software testing tool, the guidelines presented in this paper are by no means specific to AdaTEST. They are just general good practice and will provide benefit to software developers irrespective of the testing tools in use.

## 2. Guidelines

### 2.1. Modification for Testing

The most important part of the testing process occurs before the code is written, during the design phase. The aim of design for testability is to ensure that the software can be tested efficiently; ie. that thorough testing can be performed with minimum effort.

Having to modify software in order to test it presents a number of problems:

- (a) The software which is tested is no longer the software which will be used;
- (b) Modification consumes time;
- (c) Modification could introduce errors.

Any modification should be applied systematically and repeatably according to a predefined and verified scheme (such as the coverage instrumentation implemented by AdaTEST). Such a scheme will prevent the introduction of errors and minimise the risk of a modification hiding real errors in the software under test.

Where modification for testing is considered necessary, a project should develop and document a scheme as a general case for the modification which can be verified and applied systematically to all units. Implementation should also be automated, so that modification is repeatable and does not consume effort each time a unit is re-tested:

- (a) Use an edit macro to implement the general case of the modification and systematically parse all source files;
- (b) AdaTEST also provides a “TESTCODE” annotation, which can be used to mark code which should only be compiled for testing.

The aim of the designer should be to make unit specific (as opposed to systematic) modifications unnecessary, by ensuring that the software is designed to be testable from the outset. In many cases, software which requires specific modifications for testing often indicates a design which is weak from other points of view, such as modularity and portability.

## **2.2. Program Structure**

In general, the use of well known structured programming practices, such as avoidance of GOTO statements and designing subprograms, blocks, loops etc. with one entry point and controlled use of exit points will promote testability.

### **2.2.1. Unit and File Size**

Before going further into the topic of unit and file size, it is necessary to determine just what a unit is. In Ada, a unit is usually defined as a library level package or subprogram, although subprograms separate from a package body may sometimes be considered units for test purposes;

Practical test strategies are normally focused on isolation testing, supported where necessary by an element of bottom up testing (see the IPL paper “*Organisational Approaches to Unit Testing*”). In order to facilitate isolation of a unit for testing purposes, it is best to use separate source files for each testable item. Library level subprograms, package specifications and package bodies should each reside in their own individual source files.

Subprograms should be declared separate to package bodies, enabling them to be replaced with stubs where required for testing. If it is planned to test and configuration manage a separate subprogram independently of the package body, it should reside in its own separate source file. Otherwise, there is no reason other than file size why separate subprograms should not follow the package body in a single source file.

Contrary to common structured programming practice, it is best to avoid decomposition into lots of small units unless absolutely necessary. Many small units will actually contain more errors per line of code than fewer larger units.

The obvious reason for this is that interfaces are a common source of error and large numbers of small units result in large numbers of interfaces. However, in Ada interfaces are fully specified and the syntax enforced by the compiler, yet small units still have

proportionately more errors per line of code! Hypotheses on the reason for this are largely based upon cognitive psychology, suggesting that the human brain is more efficient at looking at problems of certain sizes and structures.

There are also general economic reasons for avoiding large numbers of small units. Each unit carries an overhead in documentation, management and verification, irrespective of the unit's size. This overhead will push up the cost of a system structured as a greater number of smaller units. There will be exceptions to this guideline, for example, hardware and environment dependencies should be isolated to small dedicated units (see section 2.3).

### **2.2.2. Interfaces**

Even with Ada's strong typing, complex interfaces are more error-prone to use and difficult to simulate during testing.

Interfaces between units should be simple and well specified. Keeping the number of arguments to a minimum simplifies the interface to a unit, thus reducing the cost of isolation testing both the unit itself and all users of the unit.

More interfaces (of whatever complexity) means more stubbing effort during testing, so designers should also aim to keep the number of interfaces to a minimum. Designers have to make a trade-off between fewer and larger monolithic units with complex interfaces, and many smaller units with simple interfaces.

Public data in Ada packages is effectively added to the interface of all units with visibility of the data, increasing the complexity of testing effort. In addition, units using the data will be dependant upon its declaration, providing a risk of increased re-testing effort following future modification. In general, designs should be structured so as to encapsulate global data within packages as described in 2.5.

### **2.2.3. Global Structure**

Many of the guidelines given for designing interfaces can also be generalised to more global program structuring guidelines. If a number of units comprise a functional subsystem, the functional subsystem should present a small and clean interface to software which uses the subsystem. The interface to the subsystem should ideally present a complete encapsulation of the functionality or service which the subsystem provides. In Ada, this means encapsulation of the interface within a single package which provides the subsystem's "public" interface.

In any design, there will be some units which are used throughout a system. Particular care must be taken to ensure that these units have complete, flexible and stable interfaces, which are suitable both for testing such units and for stubbing when testing other units.

### **2.2.4. Loops and Control Structures**

Deeply nested control structures (decisions within loops within decisions within decisions within loops..... etc.) create complex paths through a unit, hence making it difficult to design test cases to exercise each path and consequently leading to poor test coverage. Furthermore, the control expressions leading to paths may contain conditions which are mutually exclusive, resulting in a path which is infeasible and possibly even leading to redundant or unreachable code (which will obviously be difficult to test!).

Options available to designers are to decompose some of the processing to other units or to re-structure the software to reduce the nesting of control structures. Changing the sense of conditions can often be used to map deeply nested decisions into a sequential decisions at the same level of nesting. When redundant or unreachable code is encountered, the design should be modified and the redundant code removed.

Complex loops can be especially difficult to test. Test designers have two essentially separate problems:

- Testing the iteration scheme;
- Testing the decisions within the loop.

The solution is to consider separating a loop body from the code which iterates the loop, with the loop body becoming a separate subprogram. This enables loop bodies to be thoroughly tested without having to worry about the iteration scheme, and for the iteration scheme to be thoroughly tested without the distraction of the loop body.

The interface between the iteration scheme and the subprogram containing the loop body should then be thoroughly tested as a later part of unit testing or during integration.

Implementing a loop body in a separate subprogram must be weighed against coupling and calling overhead. In the past, the calling overhead for small loops with lots of iterations could have been a problem. However, modern Ada compilers provide “inline” optimisation of subprogram calls, both at the direction of the programmer and automatically as part of global optimisation.

### 2.2.5. *Recursion*

Recursive algorithms are in general difficult to prove. In an isolation testing strategy, how can a subprogram be isolated from calls to itself? The first solution is to avoid unnecessary recursion. In many cases there are equivalent algorithms using loops which can be structure as suggested in 2.2.4 above, providing separate test access to the iteration scheme and to the body of the algorithm.

Where recursion is essential, designers should consider splitting a subprogram into a mutually recursive pair of subprograms, even if the sole purpose of the second subprogram is to call the first. To test a subprogram which uses recursion, the other subprogram in the pair can be replaced with a stub, isolating the subprogram under test from recursion. As explained under loops in 2.2.4 above, such a subprogram need not represent a major execution overhead.

Thorough testing of the actual recursive calls between the pairs of subprograms again becomes a matter for integration testing.

## 2.3. **Environment**

There are a number of areas where software may interface to the environment, including:

- Direct access to memory mapped devices;
- Ada run-time system and pre defined library;
- Other calls to environment software, such as an operating system or real time kernel;

- Calls to third party software.

When conducting unit testing, it is usually expedient to isolate the unit under test from its environment. In the case of host-target testing, isolation from the environment is a prerequisite. The case of host-target testing is examined in detail in the IPL paper “*Host-Target Testing*”. Many of the principles of isolation from the environment presented in *Host-Target Testing* are applicable to general design for testability, even when host-target development is not an issue.

Interfaces to the environment should be isolated, removing environment dependencies from all remaining software. The term “*wrapper*” is often used to describe units which are used solely to achieve isolation from the environment.

For example, it can be impractical to stub calls to the package TEXT\_IO, but it will be much simpler to stub calls to a wrapper for TEXT\_IO.

Isolation and avoidance of environment dependencies are generally considered to be good design practice, so just designing with testability in mind can improve the quality of software.

## 2.4. Algorithms

Complex single statement algorithms or calculations should be avoided. Such algorithms and calculations should be broken up into sequences of statements which use intermediate variables at meaningful places. This will promote readability and simplify the debugging and maintenance of the algorithm.

The execution sequence of the components of the algorithm will be under the developers control. If it is convenient to replace calls to functions in the algorithm with stubs, the call sequence will be easier to predict. Dynamic analysis results, such as coverage analysis, will be clearer and easier to interpret, because the reporting will be based upon a number of simpler statements as opposed to a single complex statement.

AdaTEST assertions can be embedded at key points to verify that the algorithm or calculation is behaving as expected. It can also be advantageous to make intermediate variables within a unit visible to a test point, so that they can be examined directly from an AdaTEST script. The use of test points is described in the IPL paper “*Achieving Testability when Using Ada Packaging and Data Hiding Methods*”.

Irrespective of unit size, it is often a good idea to isolate complex calculations in units by themselves. This will enable the calculation to be rigorously tested independently from any logic which selects its execution.

## 2.5. Data

Testing software which uses access types is always a problem. When comparing two access variables, should they point to the same location, or should they point to different locations which contain the same values? This can be difficult to resolve for just one level of indirection and is a frequent source of programming error. When multiple levels of indirection are involved, the level of confusion escalates, so a general guideline is to minimise the use of access types. One solution is to fully encapsulate access types within packages as private types, thereby ensuring that all operations on the data are controlled by the package. In Ada95, controlled types can be used to add further protection to the use of access types.

Testing often comes into conflict with the principles of encapsulation and abstraction. White box testability demands access to data, whilst encapsulation and abstraction requires that the scope of data should be minimised. A solution is the use of software test points as temporary or permanent features of the software to provide test only access to hidden data. The use of test points is described in the IPL paper “*Achieving Testability when Using Ada Packaging and Data Hiding Methods*”.

The representation of data structures is frequently dependant upon compilers and the environment. The broad issue of environment dependencies is addressed in section 2.3 of this paper. In general, it is best to avoid low level manipulation of arrays, structures and records (ie. breaking data type rules provided by the language and semantics of the design). Where low level manipulation of data is unavoidable, it should be isolated in low level routines.

## 2.6. **Dynamic Memory**

The use of dynamic memory allocation and deallocation is a frequent source of errors which are dependent the length of time a system has been running. It can be difficult to design meaningful tests for dynamically allocated data. From a testability and reliability point of view, the use of dynamically allocated memory should be avoided wherever a reasonable alternative exists.

It is hard to give general rules. Designers just have to be careful with dynamic memory allocation and deallocation. Most importantly, the design must acknowledge which unit owns an item of dynamically allocated data and is responsible for deallocation and garbage collection when the item is no longer needed. Wherever possible, the scope of dynamically allocated memory should be tightly bound. For Ada software, dynamic memory is controlled through the use of access types. The general recommendations made above under “Data” provide guidance.

Even if the use of allocation and deallocation within a software application can be fully verified, there is still the risk that deallocation and garbage collection in the underlying environment is imperfect, leading to obscure ageing effects as the software executes over an extended period of time. Most commercially available operating systems and database management systems suffer from such problems to some extent.

## 2.7. **Initialisation**

A software system frequently has many initialisation and termination paths. If the software under test is initialised implicitly, a test harness will only be able to test one initialisation and one termination path. Further tests would require further test executables.

On the other hand, by avoiding implicit initialisation and handling all initialisation explicitly, a test script can explicitly re-initialise the software under test, enabling many initialisation and termination paths to be tested from a single test script.

An added advantage is that initialisation sequences are now under the developer’s explicit control, as opposed to being at the whim of the compiler and linker.

## 3. **Summary**

This paper has provided some guidelines for designing testable Ada software. Many of these guidelines represent “good engineering practice”, even if it were not for testability.

In some cases the guidelines conflict and require interpretation for individual circumstances. There are no "black and white" rules about design.

Although IPL supplies the AdaTEST software testing tool, the guidelines presented in this paper are by no means specific to AdaTEST. They are just general good testing practice and will provide benefit to software developers irrespective of the testing tools in use. Just thinking about testability will improve the structure and legibility of software, reducing the number of bugs even before the actual tests are designed.

Most importantly, how the software will be tested and design for testability are activities which must be considered early in the software lifecycle, not pushed aside until the unit testing phase is imminent.