Designing
Unit Test Cases

Executive Summary

Producing a test specification, including the design of test cases, is the level of test design which has the highest degree of creative input. Furthermore, unit test specifications will usually be produced by a large number of staff with a wide range of experience, not just a few experts.

This paper provides a general process for developing unit test specifications and then describes some specific design techniques for designing unit test cases. It serves as a tutorial for developers who are new to formal testing of software, and as a reminder of some finer points for experienced software testers.

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.

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 FM1589

Last Update:31/07/96 08:24
File: DES_UT.DOC

1. Introduction

The design of tests is subject to the same basic engineering principles as the design of software. Good design consists of a number of stages which progressively elaborate the design. Good test design consists of a number of stages which progressively elaborate the design of tests:

- Test strategy;
- Test planning;
- Test specification;
- Test procedure.

These four stages of test design apply to all levels of testing, from unit testing through to system testing. This paper concentrates on the specification of unit tests; i.e. the design of individual unit test cases within unit test specifications. A more detailed description of the four stages of test design can be found in the IPL paper "An Introduction to Software Testing".

The design of tests has to be driven by the specification of the software. For unit testing, tests are designed to verify that an individual unit implements all design decisions made in the unit's design specification. A thorough unit test specification should include positive testing, that the unit does what it is supposed to do, and also negative testing, that the unit does not do anything that it is not supposed to do.

Producing a test specification, including the design of test cases, is the level of test design which has the highest degree of creative input. Furthermore, unit test specifications will usually be produced by a large number of staff with a wide range of experience, not just a few experts.

This paper provides a general process for developing unit test specifications, and then describes some specific design techniques for designing unit test cases. It serves as a tutorial for developers who are new to formal testing of software, and as a reminder of some finer points for experienced software testers.

2. Developing Unit Test Specifications

Once a unit has been designed, the next development step is to design the unit tests. An important point here is that it is more rigorous to design the tests before the code is written. If the code was written first, it would be too tempting to test the software against what it is observed to do (which is not really testing at all), rather than against what it is specified to do.

A unit test specification comprises a sequence of unit test cases. Each unit test case should include four essential elements:

- A statement of the initial state of the unit, the starting point of the test case (this is only applicable where a unit maintains state between calls);
- The inputs to the unit, including the value of any external data read by the unit;
- What the test case actually tests, in terms of the functionality of the unit and the analysis used in the design of the test case (for example, which decisions within the unit are tested);

- The expected outcome of the test case (the expected outcome of a test case should always be defined in the test specification, prior to test execution).

The following subsections of this paper provide a six step general process for developing a unit test specification as a set of individual unit test cases. For each step of the process, suitable test case design techniques are suggested. (Note that these are only suggestions. Individual circumstances may be better served by other test case design techniques). Section 3 of this paper then describes in detail a selection of techniques which can be used within this process to help design test cases.

2.1. Step 1 - Make it Run

The purpose of the first test case in any unit test specification should be to execute the unit under test in the simplest way possible. When the tests are actually executed, knowing that at least the first unit test will execute is a good confidence boost. If it will not execute, then it is preferable to have something as simple as possible as a starting point for debugging.

Suitable techniques:

- Specification derived tests

- Equivalence partitioning

2.2. Step 2 - Positive Testing

Test cases should be designed to show that the unit under test does what it is supposed to do. The test designer should walk through the relevant specifications; each test case should test one or more statements of specification. Where more than one specification is involved, it is best to make the sequence of test cases correspond to the sequence of statements in the primary specification for the unit.

Suitable techniques:

- Specification derived tests

- Equivalence partitioning

- State-transition testing

2.3. Step 3 - Negative Testing

Existing test cases should be enhanced and further test cases should be designed to show that the software does not do anything that it is not specified to do. This step depends primarily upon error guessing, relying upon the experience of the test designer to anticipate problem areas.

Suitable techniques:

- Error guessing

- Boundary value analysis

- Internal boundary value testing

- State-transition testing

2.4. Step 4 - Special Considerations

Where appropriate, test cases should be designed to address issues such as performance, safety requirements and security requirements.

Particularly in the cases of safety and security, it can be convenient to give test cases special emphasis to facilitate security analysis or safety analysis and certification. Test cases already designed which address security issues or safety hazards should be identified in the unit test specification. Further test cases should then be added to the unit test specification to ensure that all security issues and safety hazards applicable to the unit will be fully addressed.

Suitable techniques:

- Specification derived tests

2.5. Step 5 - Coverage Tests

The test coverage likely to be achieved by the designed test cases should be visualised. Further test cases can then be added to the unit test specification to achieve specific test coverage objectives. Once coverage tests have been designed, the test procedure can be developed and the tests executed**.**

Suitable techniques:

- Branch testing

- Condition testing

- Data definition-use testing

- State-transition testing

2.6. Test Execution

A test specification designed using the above five steps should in most cases provide a thorough test for a unit. At this point the test specification can be used to develop an actual test procedure, and the test procedure used to execute the tests. For users of AdaTEST or Cantata, the test procedure will be an AdaTEST or Cantata test script.

Execution of the test procedure will identify errors in the unit which can be corrected and the unit re-tested. Dynamic analysis during execution of the test procedure will yield a measure of test coverage, indicating whether coverage objectives have been achieved. There is therefore a further coverage completion step in the process of designing test specifications.

2.7. Step 6 - Coverage Completion

Depending upon an organisation's standards for the specification of a unit, there may be no structural specification of processing within a unit other than the code itself. There are also likely to have been human errors made in the development of a test specification. Consequently, there may be complex decision conditions, loops and branches within the code for which coverage targets may not have been met when tests were executed. Where coverage objectives are not achieved, analysis must be conducted to determine why. Failure to achieve a coverage objective may be due to:

- Infeasible paths or conditions - the corrective action should be to annotate the test specification to provide a detailed justification of why the path or condition is not tested. AdaTEST provides some facilities to help exclude infeasible conditions from Boolean coverage metrics.
- Unreachable or redundant code - the corrective action will probably be to delete the offending code. It is easy to make mistakes in this analysis, particularly where defensive programming techniques have been used. If there is any doubt, defensive programming should not be deleted.
- Insufficient test cases - test cases should be refined and further test cases added to a test specification to fill the gaps in test coverage.

Ideally, the coverage completion step should be conducted without looking at the actual code. However, in practice some sight of the code may be necessary in order to achieve coverage targets. It is vital that all test designers should recognise that use of the coverage completion step should be minimised. The most effective testing will come from analysis and specification, not from experimentation and over dependence upon the coverage completion step to cover for sloppy test design.

Suitable techniques:

- Branch testing

- Condition testing

- Data definition-use testing

- State-transition testing

2.8. General Guidance

Note that the first five steps in producing a test specification can be achieved:

- Solely from design documentation;
- Without looking at the actual code;
- Prior to developing the actual test procedure.

It is usually a good idea to avoid long sequences of test cases which depend upon the outcome of preceding test cases. An error identified by a test case early in the sequence could cause secondary errors and reduce the amount of real testing achieved when the tests are executed.

The process of designing test cases, including executing them as "thought experiments", often identifies bugs before the software has even been built. It is not uncommon to find more bugs when designing tests than when executing tests.

Throughout unit test design, the primary input should be the specification documents for the unit under test. While use of actual code as an input to the test design process may be necessary in some circumstances, test designers must take care that they are not testing the code against itself. A test specification developed from the code will only prove that the code does what the code does, not that it does what it is supposed to do.

3. Test Case Design Techniques

The preceding section of this paper has provided a "recipe" for developing a unit test specification as a set of individual test cases. In this section a range of techniques which can be to help define test cases are described.

Test case design techniques can be broadly split into two main categories. Black box techniques use the interface to a unit and a description of functionality, but do not need to know how the inside of a unit is built. White box techniques make use of information about how the inside of a unit works. There are also some other techniques which do not fit into either of the above categories. Error guessing falls into this category.

Black box (functional)	**White box (structural)**	**Other**
Specification derived tests	Branch testing	Error guessing
Equivalence partitioning	Condition testing	
Boundary value analysis	Data definition-use testing	
State-transition testing	Internal boundary value testing	

Table 3.1 - Categories of Test Case Design Techniques

The most important ingredients of any test design are experience and common sense. Test designers should not let any of the given techniques obstruct the application of experience and common sense.

The selection of test case design techniques described in the following subsections is by no means exhaustive. Further information on techniques for test case design can be found in "Software Testing Techniques" 2nd Edition, B Beizer,Van Nostrand Reinhold, New York 1990.

3.1. Specification Derived Tests

As the name suggests, test cases are designed by walking through the relevant specifications. Each test case should test one or more statements of specification. It is often practical to make the sequence of test cases correspond to the sequence of statements in the specification for the unit under test. For example, consider the specification for a function to calculate the square root of a real number, shown in figure 3.1.

> Input - real number
>
> Output - real number
>
> When given an input of 0 or greater, the positive square root of the input shall be returned. When given an input of less than 0, the error message "Square root error - illegal negative input" shall be displayed and a value of 0 returned. The library routine Print_Line shall be used to display the error message.

Figure 3.1 - Functional Specification for Square Root

There are three statements in this specification, which can be addressed by two test cases. Note that the use of Print_Line conveys structural information in the specification.

Test Case 1: Input 4, Return 2

- Exercises the first statement in the specification

("When given an input of 0 or greater, the positive square root of the input shall be returned.").

Test Case 2: Input -10, Return 0, Output "Square root error - illegal negative input" using Print_Line.

- Exercises the second and third statements in the specification

("When given an input of less than 0, the error message "Square root error - illegal negative input" shall be displayed and a value of 0 returned. The library routine Print_Line shall be used to display the error message.").

Specification derived test cases can provide an excellent correspondence to the sequence of statements in the specification for the unit under test, enhancing the readability and maintainability of the test specification. However, specification derived testing is a positive test case design technique. Consequently, specification derived test cases have to be supplemented by negative test cases in order to provide a thorough unit test specification.

A variation of specification derived testing is to apply a similar technique to a security analysis, safety analysis, software hazard analysis, or other document which provides supplementary information to the unit's specification.

3.2. Equivalence Partitioning

Equivalence partitioning is a much more formalised method of test case design. It is based upon splitting the inputs and outputs of the software under test into a number of partitions, where the behaviour of the software is equivalent for any value within a particular partition. Data which forms partitions is not just routine parameters. Partitions can also be present in data accessed by the software, in time, in input and output sequence, and in state.

Equivalence partitioning assumes that all values within any individual partition are equivalent for test purposes. Test cases should therefore be designed to test one value in each partition. Consider again the square root function used in the previous example. The square root function has two input partitions and two output partitions, as shown in table 3.2.

Input Partitions		Output Partitions	
i	<0	**a**	>=0
ii	>=0	**b**	Error

Table 3.2 - Partitions for Square Root

These four partitions can be tested with two test cases:

Test Case 1: Input 4, Return 2

- Exercises the >=0 input partition (ii)
- Exercises the >=0 output partition (a)

Test Case 2: Input -10, Return 0, Output "Square root error - illegal negative input" using Print_Line.

- Exercises the <0 input partition (i)
- Exercises the "error" output partition (b)

For a function like square root, we can see that equivalence partitioning is quite simple. One test case for a positive number and a real result; and a second test case for a negative number and an error result. However, as software becomes more complex, the identification of partitions and the inter-dependencies between partitions becomes much more difficult, making it less convenient to use this technique to design test cases. Equivalence partitioning is still basically a positive test case design technique and needs to be supplemented by negative tests.

3.3. Boundary Value Analysis

Boundary value analysis uses the same analysis of partitions as equivalence partitioning. However, boundary value analysis assumes that errors are most likely to exist at the boundaries between partitions. Boundary value analysis consequently incorporates a degree of negative testing into the test design, by anticipating that errors will occur at or near the partition boundaries. Test cases are designed to exercise the software on and at either side of boundary values. Consider the two input partitions in the square root example, as illustrated by figure 3.2.

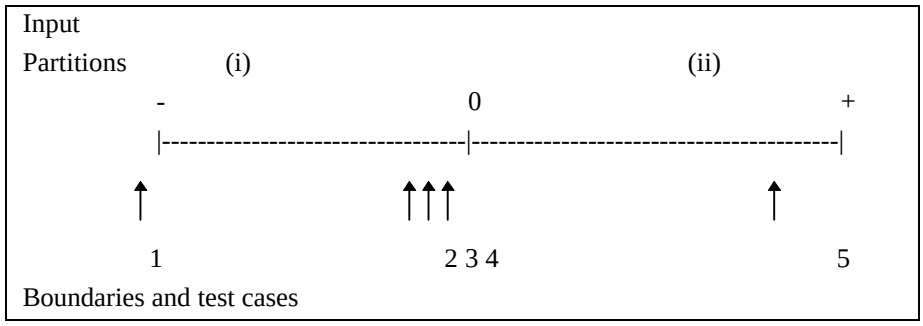

Figure 3.2 - Input Partition Boundaries in Square Root

The zero or greater partition has a boundary at 0 and a boundary at the most positive real number. The less than zero partition shares the boundary at 0 and has another boundary at the most negative real number. The output has a boundary at 0, below which it cannot go.

Test Case 1: Input {the most negative real number}, Return 0, Output "Square root error - illegal negative input" using Print_Line

- Exercises the lower boundary of partition (i).

Test Case 2: Input {just less than 0}, Return 0, Output "Square root error - illegal negative input" using Print_Line

- Exercises the upper boundary of partition (i).

Test Case 3: Input 0, Return 0

- Exercises just outside the upper boundary of partition (i), the lower boundary of partition (ii) and the lower boundary of partition (a).

Test Case 4: Input {just greater than 0}, Return {the positive square root of the input}

- Exercises just inside the lower boundary of partition (ii).

Test Case 5: Input {the most positive real number}, Return {the positive square root of the input}

- Exercises the upper boundary of partition (ii) and the upper boundary of partition (a).

As for equivalence partitioning, it can become impractical to use boundary value analysis thoroughly for more complex software. Boundary value analysis can also be meaningless for non scalar data, such as enumeration values. In the example, partition (b) does not really have boundaries. For purists, boundary value analysis requires knowledge of the underlying representation of the numbers. A more pragmatic approach is to use any small values above and below each boundary and suitably big positive and negative numbers

3.4. State-Transition Testing

State transition testing is particularly useful where either the software has been designed as a state machine or the software implements a requirement that has been modelled as a state machine. Test cases are designed to test the transitions between states by creating the events which lead to transitions.

When used with illegal combinations of states and events, test cases for negative testing can be designed using this approach. Testing state machines is addressed in detail by the IPL paper "Testing State Machines with AdaTEST and Cantata".

3.5. Branch Testing

In branch testing, test cases are designed to exercise control flow branches or decision points in a unit. This is usually aimed at achieving a target level of Decision Coverage. Given a functional specification for a unit, a "black box" form of branch testing is to "guess" where branches may be coded and to design test cases to follow the branches. However, branch testing is really a "white box" or structural test case design technique. Given a structural specification for a unit, specifying the control flow within the unit, test cases can be designed to exercise branches. Such a structural unit specification will typically include a flowchart or PDL.

Returning to the square root example, a test designer could assume that there would be a branch between the processing of valid and invalid inputs, leading to the following test cases:

Test Case 1: Input 4, Return 2

- Exercises the valid input processing branch

Test Case 2: Input -10, Return 0, Output "Square root error - illegal negative input" using Print_Line.

- Exercises the invalid input processing branch

However, there could be many different structural implementations of the square root function. The following structural specifications are all valid implementations of the square root function, but the above test cases would only achieve decision coverage of the first and third versions of the specification.

```
If input<0 THEN
        CALL Print_Line "Square root error - illegal negative input"
        RETURN 0
ELSE
        Use maths co-processor to calculate the answer
        RETURN the answer
END_IF
```

Figure 3.3(a) - Specification 1

```
If input<0 THEN
        CALL Print_Line "Square root error - illegal negative input"
        RETURN 0
ELSE
        IF input=0 THEN
                RETURN 0
        ELSE
                Use maths co-processor to calculate the answer
                RETURN the answer
        END_IF
END_IF
```

Figure 3.3(b) - Specification 2

```
Use maths co-processor to calculate the answer
Examine co-processor status registers
If status=error THEN
        CALL Print_Line "Square root error - illegal negative input"
        RETURN 0
ELSE
        RETURN the answer
END_IF
```

Figure 3.3(c) - Specification 3

```
If input<0 THEN
        CALL Print_Line "Square root error - illegal negative input"
        RETURN 0
ELSE_IF input=0 THEN
        RETURN 0
ELSE
        Calculate first approximation
        LOOP
                Calculate error
                EXIT_LOOP WHEN error<desired accuracy
                Adjust approximation
        END_LOOP
        RETURN the answer
END_IF
```

Figure 3.3(d) - Specification 4

It can be seen that branch testing works best with a structural specification for the unit. A structural unit specification will enable branch test cases to be designed to achieve decision coverage, but a purely functional unit specification could lead to coverage gaps.

One thing to beware of is that by concentrating upon branches, a test designer could loose sight of the overall functionality of a unit. It is important to always remember that it is the overall functionality of a unit that is important, and that branch testing is a means to an end, not an end in itself. Another consideration is that branch testing is based solely on the outcome of decisions. It makes no allowances for the complexity of the logic which leads to a decision.

3.6. Condition Testing

There are a range of test case design techniques which fall under the general title of condition testing, all of which endeavour to mitigate the weaknesses of branch testing when complex logical conditions are encountered. The object of condition testing is to design test cases to show that the individual components of logical conditions and combinations of the individual components are correct.

Test cases are designed to test the individual elements of logical expressions, both within branch conditions and within other expressions in a unit. As for branch testing, condition testing could be used as a "black box" technique, where the test designer makes intelligent guesses about the implementation of a functional specification for a unit. However, condition testing is more suited to "white box" test design from a structural specification for a unit.

The test cases should be targeted at achieving a condition coverage metric, such as Modified Condition Decision Coverage (available as Boolean Operand Effectiveness in AdaTEST). The IPL paper entitled "Structural Coverage Metrics" provides more detail of condition coverage metrics.

To illustrate condition testing, consider the example specification for the square root function which uses successive approximation (figure 3.3(d) - Specification 4). Suppose that the designer for the unit made a decision to limit the algorithm to a maximum of 10 iterations, on the grounds that after 10 iterations the answer would be as close as it would ever get. The PDL specification for the unit could specify an exit condition like that given in figure 3.4.

```
:
:
:
EXIT_LOOP WHEN (error<desired accuracy) or (iterations=10)
:
:
:
```

Figure 3.4 - Loop Exit Condition

If the coverage objective is Modified Condition Decision Coverage, test cases have to prove that both error<desired accuracy and iterations=10 can independently affect the outcome of the decision.

Test Case 1: 10 iterations, error>desired accuracy for all iterations.

- Both parts of the condition are false for the first 9 iterations. On the tenth iteration, the first part of the condition is false and the second part becomes true, showing that the iterations=10 part of the condition can independently affect its outcome.

Test Case 2: 2 iterations, error>=desired accuracy for the first iteration, and error<desired accuracy for the second iteration.

- Both parts of the condition are false for the first iteration. On the second iteration, the first part of the condition becomes true and the second part remains false, showing that the error<desired accuracy part of the condition can independently affect its outcome.

Condition testing works best when a structural specification for the unit is available. It provides a thorough test of complex conditions, an area of frequent programming and design error and an area which is not addressed by branch testing. As for branch testing, it is important for test designers to beware that concentrating on conditions could distract a test designer from the overall functionality of a unit.

3.7. Data Definition-Use Testing

Data definition-use testing designs test cases to test pairs of data definitions and uses. A data definition is anywhere that the value of a data item is set, and a data use is anywhere that a data item is read or used. The objective is to create test cases which will drive execution through paths between specific definitions and uses.

Like decision testing and condition testing, data definition-use testing can be used in combination with a functional specification for a unit, but is better suited to use with a structural specification for a unit.

Consider one of the earlier PDL specifications for the square root function which sent every input to the maths co-processor and used the co-processor status to determine the validity of the result. (Figure 3.3(c) - Specification 3). The first step is to list the pairs of definitions and uses. In this specification there are a number of definition-use pairs, as shown in table 3.3.

Definition		Use
1	Input to routine	By the maths co-processor
2	Co-processor status	Test for status=error
3	Error message	By Print_Line
4	RETURN 0	By the calling unit
5	Answer by co-processor	RETURN the answer
6	RETURN the answer	By the calling unit

Table 3.3 - Definition-Use pairs

These pairs of definitions and uses can then be used to design test cases. Two test cases are required to test all six of these definition-use pairs:

Test Case 1: Input 4, Return 2

- Tests definition-use pairs 1, 2, 5, 6

Test Case 2: Input -10, Return 0, Output "Square root error - illegal negative input" using Print_Line.

- Tests definition-use pairs 1, 2, 3, 4

The analysis needed to develop test cases using this design technique can also be useful for identifying problems before the tests are even executed; for example, identification of situations where data is used without having been defined. This is the sort of data flow analysis that some static analysis tool can help with. The analysis of data definition-use pairs can become very complex, even for relatively simple units. Consider what the definition-use pairs would be for the successive approximation version of square root!

It is possible to split data definition-use tests into two categories: uses which affect control flow (predicate uses) and uses which are purely computational. Refer to "Software Testing Techniques" 2nd Edition, B Beizer,Van Nostrand Reinhold, New York 1990, for a more detailed description of predicate and computational uses.

3.8. Internal Boundary Value Testing

In many cases, partitions and their boundaries can be identified from a functional specification for a unit, as described under equivalence partitioning and boundary value analysis above. However, a unit may also have internal boundary values which can only be identified from a structural specification. Consider a fragment of the successive approximation version of the square root unit specification, as shown in figure 3.5 (derived from figure 3.3(d) - Specification 4).

```
:
:
Calculate first approximation
LOOP
        Calculate error
        EXIT_LOOP WHEN error<desired accuracy
        Adjust approximation
END_LOOP
RETURN the answer
:
:
```

Figure 3.5 - Fragment of Specification 4

The calculated error can be in one of two partitions about the desired accuracy, a feature of the structural design for the unit which is not apparent from a purely functional specification. An analysis of internal boundary values yields three conditions for which test cases need to be designed.

Test Case 1: Error just greater than the desired accuracy

Test Case 2: Error equal to the desired accuracy

Test Case 3: Error just less than the desired accuracy

Internal boundary value testing can help to bring out some elusive bugs. For example, suppose "<=" had been coded instead of the specified "<". Nevertheless, internal boundary value testing is a luxury to be applied only as a final supplement to other test case design techniques.

3.9. Error Guessing

Error guessing is based mostly upon experience, with some assistance from other techniques such as boundary value analysis. Based on experience, the test designer guesses the types of errors that could occur in a particular type of software and designs test cases to uncover them. For example, if any type of resource is allocated dynamically, a good place to look for errors is in the deallocation of resources. Are all resources correctly deallocated, or are some lost as the software executes?

Error guessing by an experienced engineer is probably the single most effective method of designing tests which uncover bugs. A well placed error guess can show a bug which could easily be missed by many of the other test case design techniques presented in this paper. Conversely, in the wrong hands error guessing can be a waste of time.

To make the maximum use of available experience and to add some structure to this test case design technique, it is a good idea to build a check list of types of errors. This check list can then be used to help "guess" where errors may occur within a unit. The check list should be maintained with the benefit of experience gained in earlier unit tests, helping to improve the overall effectiveness of error guessing.

4. Conclusion

Experience has shown that a conscientious approach to unit testing will detect many bugs at a stage of the software development where they can be corrected economically. A rigorous approach to unit testing requires:

- That the design of units is documented in a specification before coding begins;
- That unit tests are designed from the specification for the unit, also preferably before coding begins;
- That the expected outcomes of unit test cases are specified in the unit test specification.

The process for developing unit test specifications presented in this paper is generic, in that it can be applied to any level of testing. Nevertheless, there will be circumstances where it has to be tailored to specific situations. Tailoring of the process and the use of test case design techniques should be documented in the overall test strategy.

Although the square root example used to illustrate the test case design techniques in fairly trivial, it does serve to show the principles behind the techniques. It is unlikely that any single test case design techniques will lead to a particularly thorough test specification. When used to complement each other through each stage of the test specification development process, the synergy of techniques can be much more effective. Nevertheless, test designers should not let any of these techniques obstruct the application of experience and common sense.

AdaTEST and Cantata can be used in association with any of the test case design techniques described in this paper. The dynamic analysis facilities of AdaTEST and Cantata provide a range of test coverage measures which can be used to help with steps 5 and 6 (coverage tests and coverage completion) of the unit test development process.