

## A Strategy for Testing C++

### Executive Summary

Compared with a procedural language such as C, the testing of C++ presents some novel problems. This paper discusses what those problems are and outlines an approach to the verification of C++ code. Some reference is made to implementing these strategies using the IPL Cantata++ tool.

### A Strategy for Testing C++

#### 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 475000*

*Fax: +44 (0) 1225 444400*

*email [ipl@iplbath.com](mailto:ipl@iplbath.com)*

*Last Update: 18/04/01 11:32*  
*File: Strategy for Testing C++*



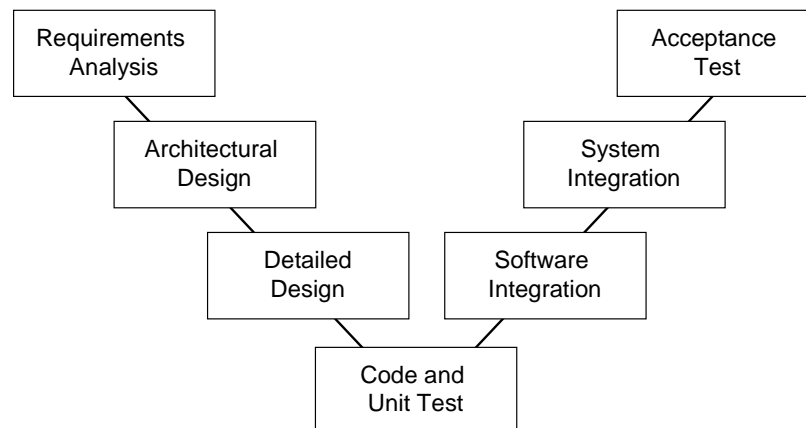
Certificate Number FM1589

This document attempts to outline a *strategy* for testing software written in C++. This means providing guidance on the thought processes you should be going through when planning verification of C++ software, preferably before any code is written. The first section is devoted to that task. The second looks at what you can do once code starts to become available for testing, though without any particular regard to any tools you might be using. Section Three describes the facilities offered by Cantata++ and indicates how this might help in selected situations.

## **1. Before Any Code is Written**

### **1.1 Decide what level to start**

The ‘ideal’ software lifecycle is very well illustrated by the so-called V-model diagram (Fig 1).



*Fig 1. The V-model of the Software Lifecycle.*

This suggests that verification of code should begin at the software unit/module level, and proceed in an orderly fashion through integration testing (various sub-levels can exist here), up to the System Acceptance Testing step, which should be the final stage before release of the software to users. This is the recommended approach in cases where the end system needs to be reasonably robust. The reliability is achieved through the different levels of testing, each providing a greater degree of confidence in the quality of the software.

However, in many situations this orderly approach is neither possible nor practical. In many situations detailed specifications of integration items and modules are not possible because the end user is unclear about what they really want. Other scenarios where it may be impossible to specify modules tightly is where performance constraints may mean that different possible implementations have to be trialed before choosing one which provides what is wanted. In situations like these it is quite likely that the only testing that takes place will be at the Acceptance Test stage.

In a properly run software development project there will need to be a degree of formality at some point in the testing side. Under these rules software is only allowed

to pass from one person to another for further testing when it can be demonstrated to do what it is supposed to do. Formality means that the test itself should be reviewable (to decide whether it means anything), the test results should give a clear statement of pass or failure (as well as meaningful diagnostics of any failures), and the test should be repeatable. (The contrast is with 'informal' testing, which is what programmers tend to like to do when no one is watching.) The big question is at what stage do you introduce formal testing?

**Start at the Unit Test level.** This essentially means testing each unit in the system, possibly building larger tests up by using previously tested units. A unit can be a module (typically C++ file) or the class itself (if more than one class per file). The minimum requirement is that a *specification* should exist, preferably in written form, describing what each unit should do. The general approach will be that having tested the base classes to get a reasonable degree of confidence that they work, each successive derived class is tested in an approach known generally as *Hierarchical Integration Testing*. More will be said about this later. This approach suits high-integrity developers because of the reliability built in by using only fully-tested components.

**Start at the Integration Test level.**

In the rush to produce a working system there may not be time to test at the unit level. In this case, a reasonable compromise is to defer formal testing to the *cluster* level, where a cluster is group of classes. In a multi-threaded application there may be advantages to testing at the *thread* level, which is usually a stage higher than clusters. Either way, or indeed whatever entry level of testing is chosen as most appropriate, there must be a specification for whatever it is you are testing.

**Start at the System Test level.**

A lot of developers choose to defer any formal test to the application level. The obvious advantage is that this gives users an early indication of what they're getting and how robust it seems. The drawback is that since the underlying components will not have been individually tested it will be unclear how robust it really is! To go some way to counter this weakness the technique of *coverage analysis* is strongly recommended as a means of finding out which parts of the overall system have not been exercised by the tests. These may typically be obscure branches of functionality or error handling. Once revealed, a decision can then be made on whether to add more tests to cover these paths. In some cases it may be decided that the unexercised paths are simply not wanted and should be removed. Much more will be said about coverage analysis later (Sections 1.4 and 2.7).

## 1.2 Design for Testability

The principle being considered under this heading is, 'what can be done to make the software testers life easier, without compromising basic OO principles such as encapsulation and data-hiding?'

After the black/white-box problem, which is discussed later (Section 1.3), the single biggest issue tends to be the need to isolate classes from each other in order that meaningful tests can be carried out when the implementations of 'external' classes are simply not available or not reliable enough to be used. In procedural languages such as C the classic approach is to stub the external modules. With C++ it is only rarely

practical to stub an external class due to the near impossibility of trying to stub constructors of the external classes (of which there may be several layers!). Add that to the fact that each class may contain many member functions, and you will quickly come to appreciate that simple stubs are not up to the job!

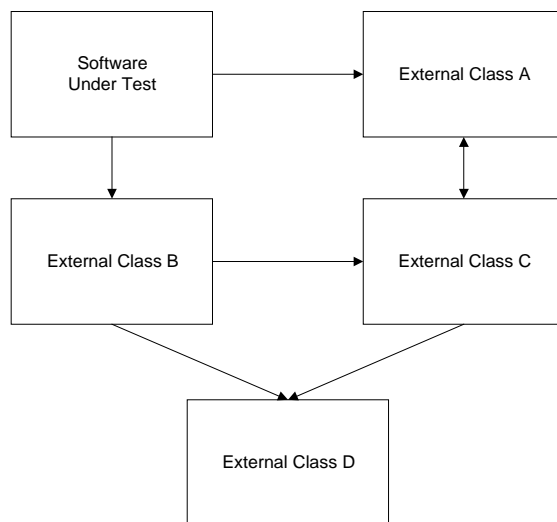
The two solutions are to either employ 'wrapping' techniques (see Section 2.4) or to make the external classes more stubbable through a design which specifies that base classes be coded as abstract classes.

Abstract Base Classes (ABCs) and their partners, Concrete Implementation Classes (CICs) form a technique for completely separating a class interface definition from its implementation. Normally a C++ class declaration defines both the interface (public) and some features of its implementation (the private part) in a single place. In the ABC/CIC technique the class is split into two:

1. The ABC which defines the (public) interface to the class as pure virtual member functions;
2. The CIC which inherits (publicly) from the ABC and provides the implementation of the class.

Clients of the class depend only on the ABC not on the CIC. To stub the class we retain the ABC but provide an alternative (stub) implementation class.

Consider the following small subsystem (arrows indicate a dependency):



*Fig 2.i Small subsystem showing unit under test and some immediate and indirect dependencies.*

Attempting to isolate the software under test would involve stubbing all the external classes, which is difficult or impossible.

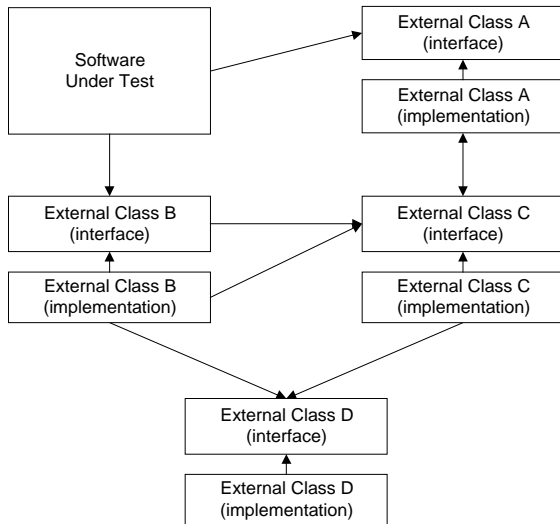


Fig 2.ii Same subsystem re-implemented using ABCs and CICs.

The subsystem can be implemented using ABCs and CICs, which looks a more complex design, but leads to easier testing.

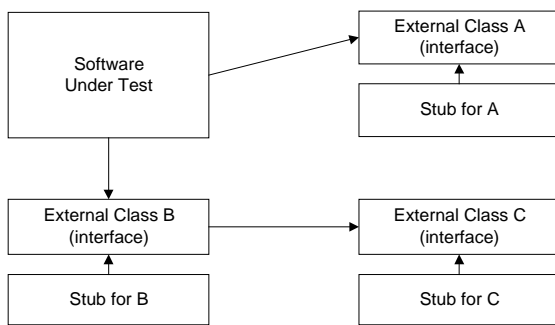


Fig 2.iii Software can now be tested with a manageable array of stub classes.

In the new design it is now possible to test the software using stubs for classes A, B and C, and omitting the stub for class D completely.

Unfortunately, there is generally speaking, a price to be paid for use of this technique, namely that each virtual member call involves an additional indirect memory access. This can lead to unacceptable levels of inefficiency, so this technique is not a panacea. A later section (2.4) describes the use of ‘wrapping’ as an alternative method of simulating external classes, but nevertheless use of ABCs does represent a useful approach for enabling the isolation of significant sub-systems. For more detail of these ideas see [Dorman#1] and [Lakos]. For more on how the ABC/CIC approach can improve maintainability see [Martin].

### Design Validation with Metrics

There is a stage in software design where the code structure has been mapped out as C++ header files, and it would be worth gaining some idea of how ‘good’ this is, in terms of OO design. This subject is covered in more detail later (Section 2.1), but for now just note that some OO metrics can be useful.

## 1.3 Code for Testability

### White-box access – ‘friend’

As hinted earlier one of the biggest questions facing the testers of C++ classes is black/white-box problem. Essentially this means deciding whether to test a class purely by its public interface (black-box testing) or whether to gain access to the private parts of the class (white-box testing) in order to increase efficiency. Typical advantages gained by the latter would be the opportunity to set or check the value of private data items, or the ability to call private member functions directly.

C++ provides the `friend` keyword to allow an external object, declared as a class, to access the private or protected parts of the class, which is the obvious way to allow white-box testing. If you are writing your own test scripts in C++ and want to use white-box techniques it will be useful to get into habit of including `friend` declarations for a test class in every class header file. More will be said about use of the C++ `friend` mechanism in Section 2.3.

### Test Re-use – Factory classes

Another useful design strategy to is Factory classes. A simplistic approach to testing objects will be to use a class definition locally in the script to create test objects of the right type. However, this is not possible to follow when trying to create reusable scripts where there is not enough information about the exact type of object to be created. This is usually the case when moving from a base class test (probably implemented through an abstract base class – see above) to a test of a derived class.

For example, referring to Fig 3:

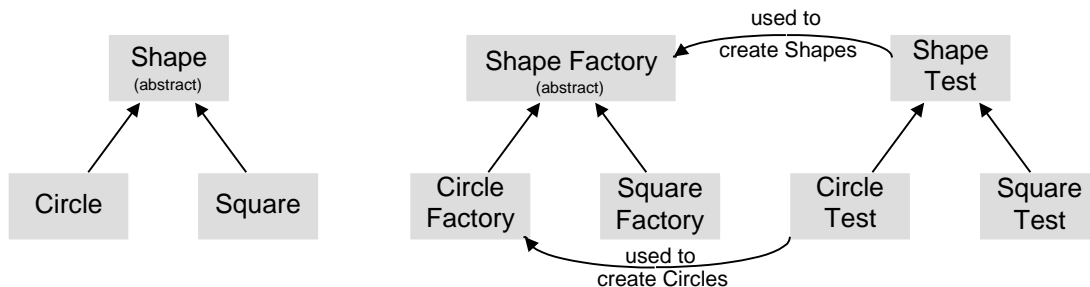


Fig 3. Shows how Factory classes and corresponding Test classes are related.

Shape is an (abstract) base class and tests can be created to check the properties of this. However, when moving to test derivations from Shape, e.g. Circle, it might be impossible to re-useably re-run the Shape test (to check the Shape properties of Circle). The solution is to use a Factory class for the creation of objects, whether for testing or for normal use, because this acts more as an interface for object creation allowing the details of the type to be supplied only when known.

Thus, a ShapeFactory class will be defined, and this can be passed (as a reference parameter) to a ShapeTest class for testing the Shape properties. Then, a CircleFactory class can be created by derivation from ShapeFactory, and CircleTest can then be created from ShapeTest. Now, when CircleTest runs the ShapeTest test cases, it passes it the CircleFactory object. This is

used by the `ShapeTest` test cases which tests that the `Circles` so created are indeed valid `Shapes`.

Use of `ShapeFactory` ensures that when a further derived class, `Square`, needs to be tested then the `Shape` tests can be re-run in the context of a `Square` object without too much difficulty.

The C++ details of `Factory` class implementation are not given here. Further information on these ideas are presented in Section 2.5.

## 1.4 How Much Testing?

We mentioned earlier that test coverage analysis is the normal method of ascertaining how well tested a unit of code has been. For a procedural language such as C you can identify a function of interest, run some test cases on this function, and then measure what proportion (usually expressed as a percentage) of the code has been executed. The general rule is that the higher the coverage achieved then the higher the confidence that it has been thoroughly tested, and correspondingly the higher the confidence in the software itself. Simple structural coverage also reveals details of the executed and un-executed code.

For example, a coverage check at the end of a test run might give the result “70% Statement Coverage achieved”. It is then up to the tester to decide whether this is considered ‘enough’ testing, and if not identify the reasons why 30% of the code has not been executed. If the reported coverage level is insufficient then more test cases must be written, or existing ones altered, to execute the untested code. Occasionally the lack of coverage may reveal unwanted or ‘dead code’, in which case it should be removed and the test re-run.

Below are outline descriptions of some of the principal coverage types applicable to C++ (and C). They are explained to help guide you on the strategic question of how much testing you should do.

### Structural Coverage

This is the easiest to understand. It is usually applied to easily identifiable code elements such as `EntryPoint`, `Statements` and `Decisions`. ‘`EntryPoint`’ coverage simply informs you that a given function or method has been called. When testing a C++ class it is reasonable to say that the absolute minimum acceptable level of coverage will be that each member function has been called at least once (i.e. 100% `EntryPoint` coverage achieved). ‘`Statements`’ includes all executable (logic rather than declarations) lines of code within a function. ‘`Decisions`’ (also referred to as `Branches`) includes constructs such as ‘`if...else...`’, ‘`switch... case...`’ and loops such as ‘`while`’ and ‘`for`’. A typical module will generally require more test cases to achieve high levels of decision coverage compared to statement coverage, but gives a correspondingly higher level of confidence that the code has been properly tested and is correct.

### Data Value Coverage

With some modules (e.g. Finite State modules) code coverage can be easy to achieve but does not really prove anything because all the functionality is contained in state tables. In these cases it is better to put test coverage confidence in the measures which

look at the values held by key variables (e.g. state). You might, for example, want to check that such a variable has held a full range of possible values before being able to say that the module has been ‘100% tested’.

The above coverage types are applicable to both procedural languages (e.g. C) and OO languages such as C++. However, for C++ it is arguable that structural coverage on its own is not enough. Described below are some examples where it is advisable to get a high degree of (structural) coverage in a range of contexts – derived classes, states, and threads.

### Context Coverage – Derived Classes

When testing derived classes it is possible to gain a misleading impression of how well an underlying base class has been tested because structural coverage achieved on the base class can accumulate. A simple example (Fig 4) illustrates this point:

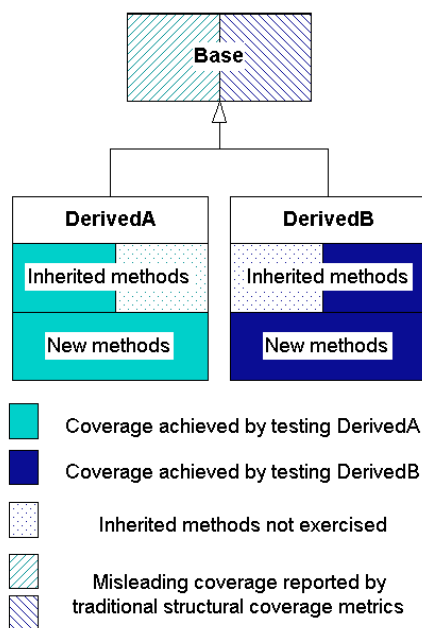


Fig 4. Shows how coverage achieved on two derived class can give a misleading impression of coverage on the common base class.

This problem applies to all the traditional structural coverage metrics – none of them take into account the need for re-test to exercise the interactions between inherited base class methods and the overridden methods in each derived class. The solution is to specify that the required level of coverage must be achieved in a specific context.

We consider this to be OO-Context Coverage [Dorman#2].

### Context Coverage – State Machines

C++ classes can frequently act as state machines i.e. the behaviour of the class depends not just on which member functions are called but also on what current ‘state’ it is in. State information is held in variables, but the actual definition of state may be a matter of interpretation. For a state machine, we may require that a structural coverage level is achieved for each member function in a range of possible states.

A simple example will hopefully illustrate the point. A stack class implementation will have at the very least a `push` and a `pop` member function. A stack can also have states 'empty', 'full' and 'partially full' (although defining these may require interpretation of private data variables). A complete test of this class should involve, at the very least, calling of both `push` and `pop` in all three possible states.

See [Binder] for more on State Machine testing.

### **Context Coverage - Threads**

Finally, the behaviour of a class may be dependent on which execution thread it is being used in. In order to avoid misleading structural coverage information it may be beneficial to generate coverage data which shows for each class in each possible thread what structural coverage levels were reached and what elements were missed. The raw coverage data can be used to analyse how the software under test actually behaves in the presence of complex thread-interaction issues.

### **Context Definition**

From the above discussions it can be seen that it may be useful to include in the private member functions of the class, a method which returns the value of the current state or the current thread ID.

## 2. Code is Written. Start Work!

### 2.1 Static Analysis Metrics

It is quite likely that detailed unit testing of code is only going to begin once a certain amount of code writing and prototyping has taken place. If that has gone well there is every likelihood that it is the prototype code itself which will be tested with a view to making it the finished product. The danger here is that the code may appear to work well, and may indeed be perfectly sound, but is it maintainable? It is a reasonable question to ask before devoting any substantial effort to detailed testing of any code module “is this worth testing?”

The factors which you need to be on the lookout for are whether the underlying design (class hierarchy or interactions) are poorly designed or over-complex, whether the implementations themselves are such that they are likely to contain a large number of faults, and indeed whether the effort needed to test the module is likely to be excessive.

Modules in procedural languages such as C are often statically analysed as part of the code review process, and complexity and other metrics are applied to estimate whether they are ‘worth testing’. The metrics in most favour for this kind of purpose were proposed by researchers such as McCabe, Halstead and others. The ‘OO’ aspects of the C++ language have tended to render the old metrics less useful, but fortunately new sets of metrics have taken their place. The popular ones include MOOSE (Metrics for OO Software Engineering), MOOD (Metrics for OO Design), and QMOOD (Quality Metrics for OO Design). Between them they define a number of metrics which can be useful for judging whether a C++ class is ‘worth testing’.

Some examples:

CODE IDENTIFY	QUALITY	TO	EXAMPLE METRICS
	Poor or Questionable Design		‘MCCABE Quality’ ‘MOOSE Lack of Cohesion among Methods’ ‘MOOD Attribute Hiding Factor’
	Estimated Number of Faults		‘MOOD Methods Defined’ ‘MOOD Attributes Defined’ ‘MOOSE Weighted Methods in Class’
	General Complexity		‘MOOSE Depth of Inheritance’ ‘QMOOD Number of Ancestors’ ‘MOOSE Number of Children’
	Estimated Test Effort		‘Methods Available’ ‘MOOSE Response for a Class’ ‘MOOSE Coupling Between Objects’ ‘MOOD Method Hiding Factor’

Table 1. Code Qualities and Metrics.

In a later section we suggest some target values for these metrics in the context of the Cantata++ tool.

## 2.2 Testing Basics - Black-Box style

As already mentioned black-box testing of an object means the verification of its behaviour purely by means of its public interface. The usual approach is to set up a series of test scenarios (or test cases) in each of which a call or series of calls are made to one or more public member functions. The checking of return values and modifiable parameters determine whether the object is doing what it is supposed to.

This is a very intuitive way to approach C++ class or cluster testing, and some further issues are discussed in the following sections:

### Constructors and Initial State of Object

Commonsense demands that testing a newly created object should first verify that that object is created in an expected initial state. If the constructor does not work then it is hard to imagine that anything else will work.

Restricting yourself to a black-box approach can make this task rather awkward because constructors are usually involved in initialising private data. This is why white-box testing (Section 2.3) is useful. Nevertheless, some constructor verification can usually be done through the public methods.

Consider a stack class. You would expect a new stack object to be created 'empty'. How are you going to check that this has occurred if you are not allowed to look inside? Here are some suggestions. Call `is_empty()` and check that it returns `true`; call `is_full()` and check that it returns `false`; call `current_size()` and check that it returns `0`; call `pop()` and see if it throws an exception. It can be argued that none of these checks individually is enough to demonstrate that the stack is empty (because they all depend on use of methods which have not themselves been independently verified), but the combined results make a reasonably convincing case.

### Functional Behaviour

Having established that a class object is created in the appropriate state you now have to set up test cases to explore whether the object does what it is supposed to do. Use this as an opportunity to try to break the object!

The standard definition of software testing can be paraphrased as, "checking that the software does what it should do, and doesn't do what it shouldn't do." Too many people (in the author's opinion) tend to stop when they have achieved the first aim, but rarely go on deliberately to stress the unit of test to the edges of its behavioural 'envelope'. Most capable programmers can write code that, under normal conditions, works correctly but may not be very robust. The point of testing is to check the hard cases as well as the easy cases.

In testing units of procedural languages like C, the techniques usually adopted for selecting test cases can include partition and boundary value analysis. Here the aim is to make calls to the functions using values deliberately chosen to stretch the unit. In C++ you are advised to do this, AND exercise ingenuity in developing test cases which reflect typical 'use cases' for the object.

For example, referring again to the stack class, it would be sensible to follow up the first test case on the initial state of the stack object, with a test case that looks at its behaviour when having items inserted and removed. The simplest example would be to push an item and then to see whether it can be popped off returning the same value. It would be sensible also to see if the stack is restored to its previous state.

A little imagination is all that is needed to construct a sequence of test cases which taken as a whole give you, the tester, a reasonable degree of confidence that the object is indeed working properly. Of course, every time a test case fails you will need to stop, debug, fix the code (or the test!), and then re-run the test which previously failed.

In devising a set of test cases it is reasonable to question whether each test case should create a new object (thus making each test case an independent item), or whether each test case should use the retained state of the object from the previous test. It is our experience that the former is generally a better approach. There may be some duplication but the benefits of having each test case independent of the others, if necessary, far outweighs the disadvantages.

### **Exception Monitoring**

C++ supports user-definable exceptions. When testing any C++ module some consideration should be given to verifying exception behaviour. There are four possibilities defined in the table below:

	Exception is Thrown	Exception is Not Thrown
Exception is Expected	Test pass	Test fail
Exception is Not Expected	Test fail	Test pass

*Table 2. Matrix to use when planning exception monitoring test cases.*

As mentioned in the previous section it should be the tester's job to search out anomalous behaviour, and verifying exceptions is definitely part of this. Since exception throwing is frequently programmed to report error returns from external calls some cleverness may need to be exercised in finding ways to force these. See the later Section (2.4) on Stubbing and Wrapping.

### **Repeatability and Portability**

Finally a couple of tips on making life easy for yourself from the start. As mentioned before, the purpose of testing is to find bugs. Once found you need to track down and 'fix' the fault, probably using a debugger. It may be that your fix does not actually work, so the test must be re-run to check for this. It may also be the case that your fix does indeed solve the initial problem but introduces a new problem. For both of these reasons it is advisable to make sure that your tests are easily repeatable. This should, ideally, be without human intervention to make them work (i.e. automated), and that where possible they should present the test results in a format that is both readable

and unambiguous. If a test fails this needs to be clear, both as a fact (i.e. ‘Test Failed’) and with symptoms provided (i.e. ‘Test Failed because...’).

A programmer working on his/her own will need repeatable tests for the reasons just mentioned. With several programmers working in a team the need for repeatability of tests is even more acute because of the possibility of changes introduced by one having a negative effect on the code written other members of the team. If a programmer changes a base class, then in addition to retesting that object every derived class will need to be retested. The way to make this practical is to ensure that each individual test is repeatable, and that the whole suite of tests can be run in an automated fashion as a regression test. During a period of intensive code development it is sensible to run the regression tests at least once a day.

Portability is another issue which may affect people. Code for a product may be developed on one platform (the development host) for eventual execution by a user on a different platform (the target). If a piece of code works perfectly correctly on the host, what guarantee is there that it will work the same on the target? Only a complete optimist will say that it must! The fact is that the C++ language definition contains many gaps and ambiguities, and furthermore that different compilers have different implementations of the same features, sometimes deliberately, sometimes by mistake. The only way to guard against compiler differences in this way is to write your tests from the outset to be portable from the host to the target, and to develop the discipline of ensuring that target tests are run!

## **2.3 Make Life Easier – Add White Box Access**

We mentioned earlier that declaring a test to be a ‘friend’ of the class that it is testing will make life easier for the tester by giving access to the private components of that class. Here are some specific examples of the benefits of doing this.

### **Being Able to Check Private Data Values**

If you make a call to a class member function (e.g. constructor) that you expect to leave the object in a new state but one with no publicly visible effects, then how do you verify that the change has occurred? As described above you can either call another public member function and so achieve verification indirectly, or you can take a direct look at the private data. Most people would argue that the direct look is ultimately more convincing and is certainly easier. Take the stack example again, and creating a new stack object which should be empty. Arguably the best verification is to look inside the stack object and check that `current_size` is 0, and that the stack head pointer is null.

### **Being Able to Set Private Data Directly**

We have previously recommended that a sequence of test cases should be devised to run independently of each other, which means creating a new, fresh, instance of your class under test for each test case. Sometimes it may be that you want to have a test case which verifies the behaviour of the object in a hard-to-reach state. One way to do this is via a sequence of public method calls which leave the object in the desired state. A quicker way is to go straight in and set that value by a simple assignment. Both approaches can have advantages and disadvantages; it is a matter of knowing what these are so that an informed decision can be taken. The biggest disadvantage is

probably the dependency of your test on the private data of the class under test. Public interfaces rarely change, once the class has been written. Private data is more likely to change, possibly forcing changes to be made to your test scripts.

### **Being Able to Call Private Methods Directly**

Private member functions exist to support the public member functions. If the private member functions do not work correctly then neither will the public ones. It can be argued that the private methods should be tested before any attempt is made to test the class at a public level, but unfortunately this is not usually easy. Having white-box access to a class implementation means that these tests can be run if necessary directly on the methods themselves. This will in turn affect how test case planning can be approached because you can now plan a set of test cases designed specifically to verify the private methods before turning attention to the behaviour of the class as a whole. Again, bear in mind the extra dependency that the test script will have on the class under test. It is a matter of weighing up the benefits, in terms of making the tests easier to write, against possible maintenance problems.

## **2.4 External Software – Simulation/Stubbing/Wrapping**

There are numerous occasions when testing a C++ unit becomes impossible without some means to simulate a piece of external software on which the unit under test depends. Two examples should illustrate this point:

- i. External class library for database services. If the unit under test depends on a third party database class, how can you check, except by indirect means, that the unit is writing correct values out? How, without elaborate initialisations, can you get the unit to read in some very specific data – possibly connecting to a test database? How can you occasionally force the library to return some specific error code to check that the unit correctly handles this?
- ii. System calls such as `operator new`. There are many occasions when the unit under test will depend, for normal operation, on a system call such as allocation of memory using the `operator new`. How can you get these system calls to return a specific value, or throw an exception in the case where you want to verify the unit's ability to handle failure of memory allocation?

The traditional mechanism, for languages such as C, of doing simulations like this has been to use a software stub for external calls. A stub is a complete replacement for the external item which contains no useful functionality itself but may have the means to check incoming values or set specific return values. Stubs work well with calls to application modules coded as functions or as abstract base classes (see Section 1.2), but will not succeed with the two examples given here, for different reasons:

- i. With external library classes it is impossible to simulate constructors especially in the depth of dependency which will typically exist in these situations. Since the source code for these libraries is not normally available, it is not an option to recode them as ABCs.
- ii. With system calls you cannot simply overwrite a system call with your own stub without causing everything else that depends on that call to fail. The traditional response is to rename the call for testing purposes and then write a stub for the renamed version. However this will not work usefully for situations where you may want the system call to work normally most

of the time, and just occasionally behave (or seem to behave) differently. It will also not work operators.

For these and other related situations you will need to consider the use of ‘wrappers’.

Fig 5 shows how a wrapper works.

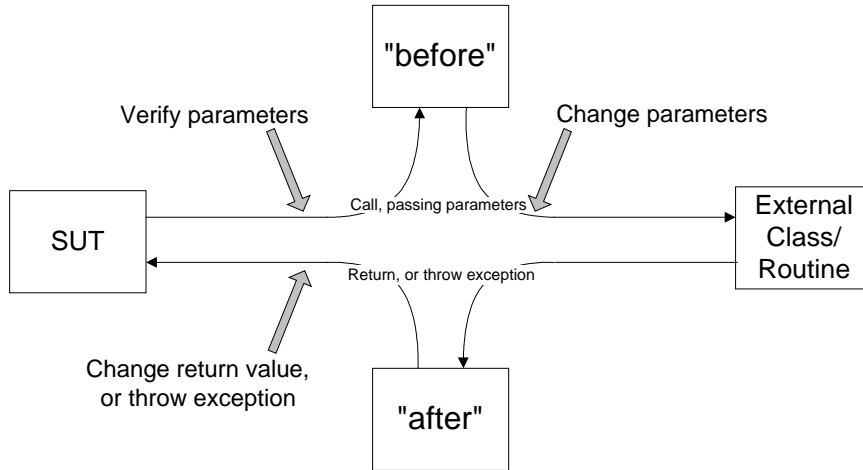


Fig 5. Wrappers – ‘Before’ and ‘After’

A wrapper is effectively a small piece of software which sits between the software under test (SUT) and the external software (class or function). It has two parts to it, namely a ‘before’ wrapper and an ‘after’ wrapper. The ‘before’ wrapper is capable of being programmed so that, when activated for a particular instance (defined by the tester) it can optionally check the parameter values being written to the external software and even change these values if wanted. The ‘after’ wrapper can, using an analogous mechanism, change a return value coming from the external software or throw an exception. The key point is that, whatever wrapper actions are programmed in the real external software is still called so that all behaviours wanted from that will actually occur.

Further useful checks can be built into wrappers, such as the ability to verify that external calls are in fact made, and made in a specific sequence as a way of verifying what is going on. A comparison between stubs and wrappers is given below (Table 3).

	Isolation Testing	Wrapping
Check call order	✓	✓ (optional)
Check parameters	✓ (optional)	✓ (optional)
Call original function	✗	✓
Set return value	✓	✓ (optional)
Throw exception	✓ (optional)	✓ (optional)
Change output parameters	✓ (optional)	✓ (optional)
Call original function with modified parameters	✗	✓

Use with system calls	✘	✓
Use selectively (based on call-site, as well as function called)	✘	✓
Original function is linked with test	✘	✓

*Table 3. Comparison between Isolation Testing with stubs and use of Wrappers.*

## 2.5 Move from Base to Derived Class

So far we have been tending to think generally of base classes and their testing. Now we consider what needs to be done when you start testing derived classes.

A naïve view might be that you only have to test the added or modified functionality which the derived class is bringing to the scene. However, it has been amply demonstrated [Harrald et al] that it is dangerous to assume that ‘old’ functionality has not been affected by use of the base class in the new context. An example might be the changed behaviour of an inherited member function if it calls a virtual member function which has been overridden in the derived class.

The safe approach in all such situations is to assume that the base class functionality may have been disturbed and to incorporate a re-run of the base class test into the derived class test. This action has the additional benefit of automatically testing the design for conformance to the Liskov Substitution Principle (LSP). The LSP is an important OO design principle which helps ensure that inheritance hierarchies are well-defined. See [Liskov] for more on this. The use of a parallel inheritance hierarchy also forms the basis of the PACT method described in [McGregor].

## 2.6 Templates

Templates are another case where code reuse promises endless productivity gains. True, but only if the specialisations are testable, and we must once again recognise the possibility that functionality may be disrupted if template specialisations are made on inappropriate types. This means of course that each specialisation needs to be individually tested. There are several ways of doing this, but the efficient way to achieve this is to write a template test which is itself a template. The script can then, with very little effort, be re-instantiated for every specialisation type.

## 2.7 Coverage Analysis.

In the earlier section (1.4) we described the various coverage types which should be considered. It was noted that structural coverage metrics define what can easily be measured, but this needs to be further considered in terms of the (OO) context in which the unit is being used.

This means, that when defining the acceptable coverage level you need to consider the following

- which unit(s) you are interested in (all class methods or a selected set)
- which structural coverage metrics you want
- which contexts you are interested in

- what level (%) you want to reach

You can, if you wish, take a very lax view of context coverage, meaning you don't care how a unit achieves structural coverage, or you can take a strict view, which means that a high level of structural coverage must be achieved for every possible context. Do not forget that State and Thread contexts are user-defined.

## 2.8 Summary So Far

A programme of code verification testing activities should be established at the very outset of a C++ development. This can even include testing during a prototype phase. The main consideration will be at which level 'formal' testing should be introduced.

There are a number of design and coding practices which, if introduced early, will make life easier for everyone concerned. These should be given consideration as soon as detailed design starts.

The amount of testing at the various stages should be considered, and specified in terms of coverage types and levels (%). The aim should be to set these to deliver a desirable level of confidence in the delivered code items, but with an eye on the cost i.e. appropriate to the integrity level and phase of the overall project.

When considering whether a given unit is 'worth testing', make use of static analysis of the source files to aid reviews. Reject any modules that fall significantly short of agreed standards.

While black-box techniques for testing classes are the obvious approach, significant productivity advantages can be gained by using white-box techniques.

The general approach to testing C++ from the base-class level up is to use 'hierarchical integration'. Nevertheless, some units may be impossible to test without use of techniques (stubbing and/or wrapping) to simulate the behaviour of external functions, classes or sub-systems. Be familiar with what these techniques can offer, and be prepared to use them when either necessity arises or convenience suggests.

### **3. How does Cantata++ help?**

If you have read and agreed with this paper so far you may hopefully be saying that these ideas represent commonsense and good practice, but how are you going to follow them without some aids? Help is at hand, in the form of the IPL Cantata++ toolset. Full product information is available at IPL, but a short summary follows.

#### **3.1 Static Analysis and Metrics**

Cantata++ support the generation of a large number of source code metrics from source code files. It can be appropriate to use these at both the design/header or the implementation stages. Metrics data is generated in the form of CSV files, which can be read into and analysed by a number of proprietary tools, such as the Excel™ spreadsheet.

#### **3.2 Dynamic Testing**

Cantata++ provides a rich set of facilities for doing dynamic testing of C++ from the base class level up. Using Cantata++ test scripts are written as C++ classes with an associated `main()` and used as ‘drivers’. Test directives are implemented as macros, and provide the means to structure a script, perform automated checks on values, monitor exceptions, and stub simple external functions. These provide a rich capability for black-box testing of classes.

Test building consists of compiling both the software under test and the Cantata++ test script(s), then linking the objects with the Cantata++ libraries (and any other specified libraries) to form a test executable. The test executable is then run. Normally Cantata++ is used in its ‘compiler driver tool’ mode, whether on PC or Unix. This has the effect of making Cantata++ as easy to use as the compiler. It provides easy access to the large number of Cantata++ options.

For white-box testing, you can automatically implement the ‘friends’ technique through the Cantata++ ‘testability instrumentation’ option (also known as source modification). This creates a copy of the original software under test, but adding the identity of the associated test script class as a ‘friend’ of the class under test.

The techniques mentioned in earlier section on use of Abstract Base classes and Factory classes are not essential for use of Cantata++ but they do help with things like test re-use and simulation of external classes. For Wrapping of external calls and classes, Cantata++ implements a variation of the ‘testability instrumentation’ option specifically for this purpose. It works by inserting wrapper calls into (a copy of) the software under test, and also by creating a ‘wrapper function template’ file, which is linked into the test executable, and which can be programmed to define the wrapper’s behaviour according to the user’s needs.

#### **3.3 Coverage Analysis**

Cantata++ implements the full range of coverage analysis possibilities mentioned in this article, as well as a few more which haven’t been described (‘MC/DC’ and ‘Boundary Value’ coverage). For all the main structural coverage types mentioned, you can overlay an OO Context coverage requirement, whether for derived classes,

templates, states or threads. For the latter two contexts, the user is required to define the states or threads which will be of interest.

Two actions are necessary to make use of Cantata++ coverage. First, you need to 'gather' the coverage data. This is achieved by using Cantata++ to instrument (a copy of) the source files containing the software under test. Secondly you need to 'report' on the coverage values obtained. You will need to add coverage checks and report directives to the test script.

Note: Coverage reporting does not have to be done at the same time as gathering coverage data. If there are no coverage directives, then the coverage data will be automatically exported to a file, to allow it be checked and reported on at a later stage. This is particularly useful for application coverage as a standalone activity.

When used normally the tool is able to generate a test report which indicates (pass, fail or warning) on whether the desired coverage level was reached. It can also produce reports which show which parts of the code (in which context) were not executed. It is then up to the user to analyse these results to decide whether more testing is needed, or possibly that some redundant code has been discovered and should be removed!

Cantata++ coverage can be applied in two ways:

#### **As Part of Unit/Integration Testing**

In this, a Cantata++ test script as described above forms the main test drive mechanism, and the coverage results contribute to and form part of the overall result picture. Thus, you could have a test which passes on logical behaviour but fails coverage, or vice versa, or indeed any combination.

#### **As a Standalone Activity**

In this, the test driving is done by some means independent of Cantata++, but the tool is nevertheless involved to provide the coverage. This is a typical situation when using a GUI tester to drive an application level test, but doing coverage to check how thorough that test is.

## **5. References**

[Binder] R. Binder, *The FREE Approach to Testing Object-Oriented Software*, <http://www.rbsc.com/pages/FREE.html>

[Dorman#1] M. Dorman, *C++ It's Testing Jim But Not As We Know It*, Proceedings of the Fifth European Congress of Software Testing and Review, 1997, paper available at IPL, <http://www.iplbath.com>

[Dorman#2] M. Dorman, *Advanced Coverage Metrics for Object-Oriented Software*, paper available at IPL, <http://www.iplbath.com>

[Harrold] M.J. Harrold, J.D. McGregor, and K.J. Fitzpatrick, *Incremental Testing of Object-Oriented Class Structures*, Proceedings of the Fourteenth International Conference on Software Engineering, 1992, pp. 68 - 80.

[IPL] Cantata++ product information at <http://www.iplbath.com>

[Lakos] J. Lakos, *Large Scale C++ Software Design*, 3 part series starting in C++ Report June 1996

[Liskov] Liskov, B and J.Wing, *A Behavioural Notion of Subtyping*, ACM Transactions on Programming Languages and Systems, Vol 16 No 6, November 1994, pages 1811-1841.

[Martin] R.C. Martin, *The Dependency Inversion Principle*, C++ Report May 1996 (also available at <http://www.oma.com>)

[McGregor] McGregor, J.D. and A.Kare, *PACT: An Architecture for Object-Oriented Component Testing*, Proceedings of the Ninth International Software Quality Week, May 1996.