

*This paper was presented at EuroSTAR'97 in Edinburgh 24-28 November 1997. The full reference to it is: Dorman, M.N., "C++ - "It's Testing, Jim, But Not As We Know It"", Proceedings of the Fifth European Conference in Software Testing, Analysis and Review, Edinburgh, Scotland, November 1997. (see <http://www.sqe.com/euro/eurhome.html> for further details).*

# **C++ "It's Testing, Jim, But Not As We Know It"**

Misha Dorman

IPL Information Processing Limited  
Eveleigh House  
Grove Street  
BATH BA1 5LR

[mishad@iplbath.com](mailto:mishad@iplbath.com)    <http://www.iplbath.com>

## **Abstract**

Object-oriented technologies, and the C++ language in particular, have enjoyed great popularity in recent years. Proponents have promised many benefits, including reduced lifecycle costs, increased reliability and improved maintainability. Unfortunately, not all of these benefits have been fully realised. One area where OO methods are relatively immature is testability and testing.

The naïve application of traditional testing techniques to object-oriented C++ software has not been completely successful. Isolation testing – a technique successfully used to “divide and conquer” traditional testing problems – becomes unmanageable when applied to OO software systems.

This paper explores the problems from a practical point of view, and discusses an improved integration testing approach based on an extension to the widely used source code instrumentation technique. This approach provides many of the advantages of isolation testing without forcing the tester to laboriously write simulated code (“stubs”) for the rest of the system.

Measuring test effectiveness of OO software using traditional coverage metrics can be unreliable and misleading. This paper discusses some metrics which provide more insight into the dynamic complexity of the software under test. In particular, they attempt to measure the extent to which the polymorphic features of the software have been exercised. In addition, modifications are proposed to the traditional coverage metrics, to make them more appropriate to C++ software.

By combining the enhanced integration testing techniques and new coverage metrics described in this paper, together with appropriate tool support, C++ testing will be more efficient and more effective.

## 1. Testing As We Know It

A vast amount of experience has been gained in the testing of software systems. Most of this experience applies to software written using procedural languages (C, Ada83), using top-down design methods.

Typically, the focus of testing is dynamic testing – the software under test is executed, and the actual behaviour compared with the expected behaviour. A dynamic test harness is used to drive and control the test: it sets up the input data required for the test, invokes the software under test, and compares the resulting output data with the expected values. Dynamic testing is performed at all levels from unit, through integration, to system-level testing.

During unit testing, isolation techniques are often used to “divide and conquer” large testing problems. Through the use of stubs, individual units (typically functions) are tested in isolation from the rest of the system. Isolation testing allows units to be tested before they are integrated with the rest of the system<sup>1</sup>. When units are subsequently integration tested, test effort can be concentrated on the correct operation of the interfaces between units. The use of stubs (which form part of the test harness) also has benefits during maintenance: re-testing at the unit level is limited to those units which have actually changed. It is not necessary to re-test all the code which depends on the changed units.

At all levels of testing, but particularly at the unit and integration levels, structural coverage analysis methods are used to measure the quality of the test. During unit testing, statement, decision and condition coverage metrics are used to ensure that all parts of the code have been exercised. During integration testing, entry-point and call-pair coverage metrics are used to ensure that the interactions between units have been fully exercised.

## 2. Why Is Testing C++ Harder?

The C++ language is considered by many to be “merely” an extension to the C language. Some practitioners have claimed that testing C++ is no different from testing C - simply redefine the unit from function to class and carry on as normal. It is true that the traditional approaches to test case design (error guessing, domain coverage, state-based testing and others) are still useful and valid. However, attempts to apply traditional testing approaches to C++ software have encountered significant difficulties. Some of these difficulties stem from the use of object-oriented design methods; others from intrinsic features of the C++ language.

### 2.1 Too Many Dependencies

Typical object-oriented designs consist of a large number of interacting components, each relatively small by itself. This is an important strength of OOD – it provides the power to break down large, complex problems into smaller ones.

An unavoidable consequence of such designs is a massive increase in the number of dependencies between units. These dependencies include containment, inheritance, reference, parameters, and return values.

Attempting to use isolation techniques during unit testing of most OO systems is bound to fail. The overhead involved in the creation of test stubs for every dependency is simply too great.

---

<sup>1</sup> even before the rest of the system is *written*. This allows much greater flexibility in the allocation of work; it is not necessary to complete coding before unit testing begins.

As if these problems were not enough, features of the C++ language create even more dependencies. Class implementation details are exposed in header files. Although clients are prevented from taking advantage of this, it causes significant problems during isolation testing.

In order to stub a constructor for a class, it is necessary to provide initialisation values for all data members of the class<sup>2</sup> - including the private ones<sup>3</sup>. However, the stub is not part of the class (it is part of the test for some other class), and as such should have no knowledge of, nor access to, the implementation details.

Remember, one of the reasons for isolation testing was to reduce the re-testing effort during maintenance. When constructors are stubbed, this benefit is lost: changes to thousands of stubs, in the unit tests for hundreds of different classes, may be required whenever the implementation of a class is changed. Even if the class didn't have any problematic data members when first written, they could be added at any time during maintenance.

Clearly, this situation is unacceptable – we have been forced to forego data hiding and abstraction. With them goes one of the main benefits of OO, the ability to change the implementation of a unit without requiring changes to its clients.

## **2.2 Integration Testing**

We have seen above that isolation testing is inappropriate for most OOD/C++ systems. The alternative to isolation testing is bottom-up testing. First the lowest level units are tested (those that have no dependencies on other units, and therefore do not require isolation). Then, the units which directly depend on the lowest level units are tested in integration with the already tested units. In this manner, testing proceeds up the dependency hierarchy, until the complete system has been integrated and tested.

At each level of bottom-up testing, the test harness must target both the low-level structural faults normally associated with a unit-level isolation test, and the interaction faults normally targeted by an integration test. This combination makes bottom-up testing more complex than either unit or integration testing alone.

The increased size of the “lump” which is the software under test makes it difficult to exercise all parts of the unit's code. The influence that the test harness has over the software under test is limited to defining an execution environment and passing in parameters. As testing proceeds to higher levels, the “already tested” units start to get in the way of the unit testing of the highest level units – sanitising the input environment and making it impossible to test obscure cases or error handling code.

In typical C++ systems, these problems are exacerbated by the use of data hiding and the presence of circular dependencies between units. Data hiding makes it even more difficult for the test harness to force the execution of “difficult” cases: it is prevented from setting the private data, or calling the private member functions, of the class under test. Circular dependencies force units to be integrated in groups instead of individually, making effective unit testing even harder to achieve.

---

<sup>2</sup> More accurately, all the data members which are of class type, and which do not have default constructors. There is often at least one such member - and as we shall see, even one is too many.

<sup>3</sup> Of course, usually all the data members are private.

## 2.3 Re-use

Much of the hype surrounding object-oriented development has focused on “re-use”. This overloaded term can apply to the re-use of code, designs, or architectures; here we are concerned primarily with the re-use of code.

Code re-use predates OOD, of course. Every time a programmer uses a library function, they are “re-using” code. Such re-use is limited, however, to that which is available in libraries. What is different about re-use in OO systems is that code is re-used in a way that provides different or additional functionality, without duplicating the common functionality. This provides the flexibility which enables large-scale re-use of components. In C++, re-use is provided through inheritance and template instantiation.

When a class inherits a member function from its base class, it is re-used in a new context. The behaviour of the inherited member can change, for example if it calls a virtual member function which has been overridden in the derived class.

When a template is instantiated on a new type, it is also being re-used in a new context. The behaviour of the template can change because its use of the operation “+”, for example, means different things when applied to different types.

Re-use through inheritance is a dynamic, run-time feature. The behaviour of the inherited member function depends on the actual (dynamic) type of the object to which it is applied. Re-use through template instantiation, on the other hand, is a compile-time feature<sup>4</sup>. The behaviour of the template depends on the static (compile-time) type on which it is instantiated.

In both cases, the behaviour of the re-used code is (potentially) different in each new context. In [Harrold], criteria were defined for the re-testing of re-used software. In essence, re-testing is necessary if the behaviour of the re-used component is changed<sup>5</sup> in the new context.

Every time code is re-used instead of written from scratch, effort is saved. However, testing may account for 30-50% of development effort. If a new test script is required whenever the unit is re-used in a new context, then the savings obtained through re-use of code will be significantly reduced.

## 2.4 Coverage

Typical OO systems consist of a large number of relatively small member functions. Applying traditional coverage metrics to these systems is certainly possible, and necessary. However, achieving 100% decision and condition coverage during a class test is usually easier than achieving the same level of coverage on the equivalent system written in a procedural language,

In an object-oriented system, an important part of the control flow arises from polymorphism (virtual functions in C++). In an equivalent procedural system, this control flow would probably have been written as explicit `if` or `switch` decisions - and would be targeted by traditional structural coverage techniques. In the OO system, this control flow is hidden from the coverage analysis – because the definitions of the coverage metrics are too restrictive.

As well as “missing” polymorphic control flow, traditional coverage metrics fail to take advantage of design information made explicit in OOD which can enable more advanced coverage analysis. In particular, many classes can be modelled as state machines. The

---

<sup>4</sup> In typical C++ implementations, re-use via templates results in duplication of object code. However, this is purely a compiler issue - the important thing is that the source code is re-used, unchanged, and automatically.

<sup>5</sup> Or potentially changed.

potential states in which an object can be, are as important a structural characteristic as the potential branches that can be taken in a member function, and as such should be measured as part of coverage analysis.

Clearly, new coverage metrics are required if we are to benefit from structural coverage analysis techniques.

### 3. Solutions

Now that we know the problems we face - what can we do about them? The techniques described below tackle each problem in turn.

#### 3.1 Design For Testability

The first problem is the inappropriateness of isolation testing for C++ systems. Remember, this is due to an increase in the number of stubs required (making isolation more expensive), and problems with stubbing constructors (making isolation impossible without breaking encapsulation). The increase in the number of dependencies is in many cases an unavoidable result of OO design. However, consideration of dependencies during design can help avoid an unnecessary increase in the dependencies between components – thus reducing testing effort (see [Lakos] for more details).

To solve the isolation problem, consider what sorts of classes *could* be stubbed: classes with no (private) data members. Abstract Base Classes (ABCs) are a perfect example of stubbable classes.

ABCs and their partners, Concrete Implementation Classes (CICs) form a technique for completely separating the class interface from its implementation. Normally, a C++ class declaration defines both the interface (the public part) and some features of the implementation (the private part) in a single place. As we have seen, this causes problems when we attempt to stub these classes. In the ABC/CIC technique, the class is split into two classes:

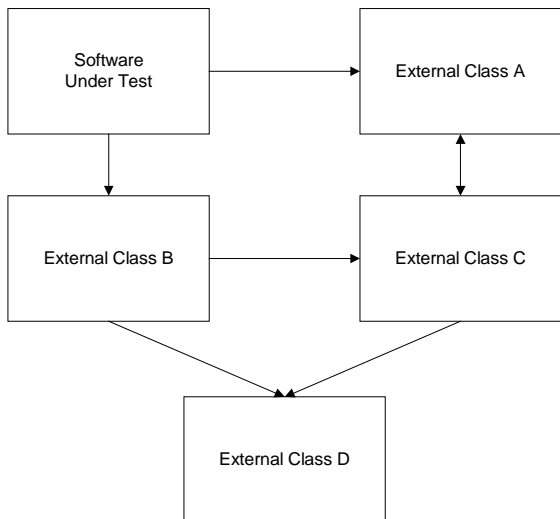
1. the ABC which defines the (public) interface to the class (as pure virtual member functions);
2. the CIC which inherits (publicly) from the ABC, and provides the implementation of the class<sup>6</sup>.

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

---

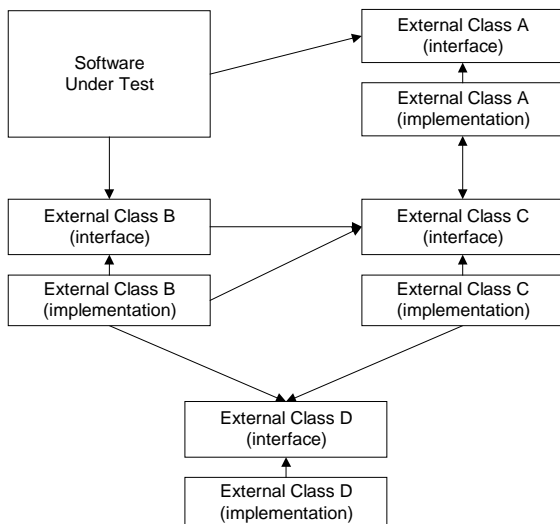
<sup>6</sup> By implementing (overriding) the pure virtual functions declared in the ABC.

Consider the following small sub-system (arrows indicate a dependency):

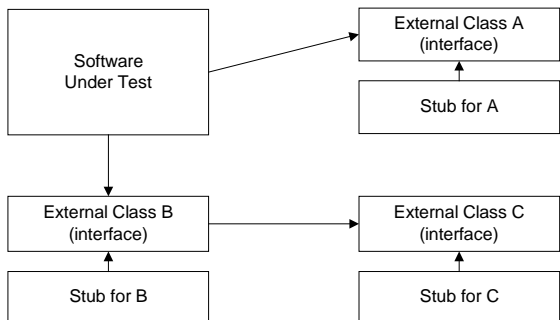


Attempting to isolate the software under test would involve stubbing all the external classes – which we have seen is difficult or impossible.

Now consider what happens if we use ABCs to separate the interface and implementation for each of the external classes:



On the surface, this design looks *more* complex – after all there are more classes! But consider how we would test the class with this design. We can stub the external classes A, B and C, and external class D can be omitted from the test altogether (it is only depended on by the implementations of B and C, which have been stubbed):



Note that in this example we have omitted the details of how objects are created. Clearly, the code which *creates* the object will depend on the implementation class (whereas code which only *uses* the object depends only on the interface class). The usual technique for managing these dependencies is to use the “Factory Method” pattern.

This technique looks to be the answer to our isolation testing problems.<sup>7</sup> Unfortunately, it is not.

There is an overhead in the use of the virtual member functions – each virtual member function call involves an additional indirect memory access<sup>8</sup>. This overhead would be unacceptable in many systems, if applied throughout. There is a trade-off to be made between the goals of efficiency, testability and maintainability.<sup>9</sup>

In addition, as testers we are often not able to influence the design of the system. The testing phase is considered by some to start after design is complete, rather than as an essential part of the complete software development process.

In the real world, the most we can hope for is that ABCs are used to isolate significant sub-systems. Within those subsystems, though, we shall still be restricted to bottom-up integration testing techniques.

### **3.2 Instrumentation for Testing**

We have resigned ourselves to using bottom-up integration testing, for at least some of our class (unit-level) testing. Some sub-systems will be stubbed, but we will be integrating with a number of (already tested) units.

It is difficult to perform unit-level testing during an integration test, for the reasons described above. Since we will be concentrating much more on integration testing, we want to make it as painless as possible.

The problems we expect during bottom-up testing are due to the data and behaviour of the software under test being hidden from the test harness – either by the data hiding features of the language, or by the processing performed by the other units which are linked with the test.

The first stage towards making integration testing easier is to allow the test harness access to the implementation details of the class under test, through a `friend` declaration. This allows the test harness to:

1. verify the correct implementation of the class by examination of its private data;
2. call private (helper) functions of the class to ensure that they are fully tested;
3. initialise private data of the class to force particular execution paths.

The second stage is to allow the test harness access to the calls made from the class under test to the other linked-in units. This is achieved through “call interface instrumentation”, an extension to the commonly used source code instrumentation technique used for coverage analysis.

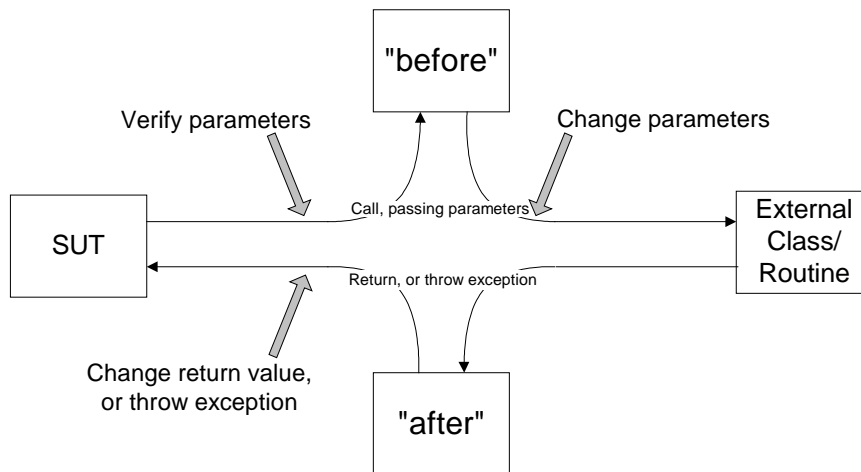
---

<sup>7</sup> It also has re-use and maintenance benefits: new implementation classes can be added to the system without changing client code. See [Martin] for more on this.

<sup>8</sup> Virtual function calls have also been shown to reduce the effectiveness of modern super-pipelined CPUs, further impacting performance.

<sup>9</sup> An alternative solution is to hide implementation data behind a `void* pointer_to_impl_data` (rather than behind an abstract class interface). This approach suffers from the same efficiency and overhead problems as the use of ABCs.

Call-back functions are called immediately **before** and **after** the original call; the call-backs “wrap” the original call:



Wrapping gives the test harness access to:

1. the order that calls are made;
2. the parameters passed to each call;
3. the return value from the linked-in function.

In addition, the test harness is able (through the call-back mechanism) to:

1. modify the return value passed back to the software under test;
2. throw an exception instead of returning a value;
3. modify any “output” parameters that are passed back to the software under test;
4. modify any other parameters before they are passed to the linked-in function.

Wrapping makes it easy to force “special cases”, such as error conditions, in order to test all parts of the class under test. For example, to verify that the software under test correctly handles the possibility of an exception thrown by a 3<sup>rd</sup> party database access library might be impossible using normal bottom-up testing techniques – how would we force the library to throw the exception?

Using wrapping, such as test is simple to implement. The appropriate call to the library is wrapped, and the call-back functions cause an exception to be thrown. This exception is then propagated to the software under test, as if it came directly from the database library.

Call interface instrumentation is in many ways similar to coverage instrumentation. Both techniques are implemented using source code instrumentation, whereby instrumentation code is added to the source code to provide the test harness with information about the execution of the software under test which would normally be hidden.

Coverage instrumentation provides the test harness with information on the proportion of functions, statements, decisions and conditions which have been exercised during the test. The instrumentation code records the execution of each statement (or decision, condition etc.) as it happens.

Call interface instrumentation provides the test harness with information on the exact calls which have been made, the order in which they occurred, and the parameters passed to each call. The instrumentation code records the calls made, and the parameters passed.

Wrapping, in combination with a bottom-up test strategy, provides many of the benefits of isolation testing, with increased flexibility, and without the scalability and maintenance problems which are associated with stubbing C++ classes. The following table compares the features of isolation testing and wrapping:

	Isolation Testing	Wrapping
Check call order	✓ <sup>i</sup>	✓ (optional)
Check parameters	✓ (optional)	✓ (optional)
Call original function	✗ <sup>ii</sup>	✓
Set return value	✓	✓ (optional)
Throw exception	✓ (optional)	✓ (optional)
Change output parameters	✓ (optional)	✓ (optional)
Call original function with modified parameters	✗	✓
Use with system calls	✗ <sup>iii</sup>	✓
Use selectively (based on call-site, as well as function called)	✗	✓
Original function is linked with test	✗	✓

<sup>i</sup> Because a stub must always provide a return value, and cannot call the original function, it must always know “where we are in the test” in order to create the correct return value.

<sup>ii</sup> A stub function is simply a replacement implementation for the original function. The two cannot be linked together in the same test harness, since they have exactly the same linkage name.

<sup>iii</sup> In general, isolation testing cannot be used with system calls. Consider a test harness which stubbed the exit() function: how would it terminate?

### 3.3 Re-use of test cases

Whenever we re-use a software component in a context which requires re-testing, we should also re-use the corresponding test cases. This re-use will be done not just at the test planning level, but through direct re-use of the test case code. Designing and implementing re-usable test cases will require more effort to make them re-usable, in the same way that creating re-usable components is harder than creating single-purpose components. In both cases, the pay-off for the initial investment comes later, when the component is re-used, either elsewhere in the system, or in other systems.

#### 3.3.1 When to Re-use?

The usual example of re-use in C++ is through the inheritance mechanism. In particular, the use of public inheritance implies the existence of an “isA” relationship between the classes. Of course, the compiler cannot enforce the rule that public inheritance is used only when there is an “isA” relationship – the compiler has no knowledge of the semantics of the classes. However, using public inheritance when there is no such relationship is asking for trouble. Any function which is declared with a Base& (or const Base&) parameter may in fact be passed a reference to a Derived object. If Derived “is-not-A” Base, then the function is likely to behave unexpectedly (i.e. wrongly).

By re-using the base class test cases when testing the derived class(es), we can verify that the “isA” relationship holds true, and that the derived class correctly implements the interface defined by the base class. This approach can even be used for abstract base classes: test cases

can be written which verify the correct implementation of the interface provided by the abstract base.

The above applies equally to re-use through template instantiation. Each template makes certain assumptions about the types on which it will be instantiated: that they are copyable, assignable, have an equality operation etc. These assumptions are both syntactic (`obj1 == obj2` is legal) and semantic (`==` defines an equivalence relation). Unfortunately, there is no way to make these assumptions explicit in the C++ language<sup>10</sup>. To verify that the assumptions have not been violated, write a set of re-usable test cases for the template, and use those test cases to test each instantiation.

Re-usable test cases will in general be behavioural tests, based on the externally visible functionality of the class under test. It is also possible to re-use structural test cases, where the code in question has not been replaced (i.e. for inherited members, or templates for which no specialisation has been defined).

### 3.3.2 Factory Classes

Writing a re-usable test case is, in most respects the same as writing a normal test case. Extra care is needed to ensure that no unwarranted assumptions are made about the types of objects.

For inheritance re-use, the test script must use references or pointers instead of simple objects. For template re-use, the test script itself is a template, parameterised by the same types as the template under test.

Most test cases create one or more objects, in preparation for the test proper. How can a re-usable test script create objects, without knowing the type of the objects being tested. The solution, again, is to use the Factory Method pattern.

The test case is passed a Factory object as a parameter, and uses the factory to create objects as needed. A different Factory class is written for each class which is to be tested. Whenever the test case is used to test a class, it is passed the corresponding Factory object.

To ensure that different Factory objects can be substituted as necessary, the Factory classes form an inheritance hierarchy of their own, which precisely mirrors that of the classes to be tested. This parallel inheritance hierarchy means that if a `Circle` “isA” `Shape`, then a `CircleFactory` “isA” `ShapeFactory`.

In the following diagram, `Shape` is an abstract class, and `Circle` is derived (publicly) from `Shape`. `ShapeTest` contains the test cases which test for “Shape-ness”. `CircleTest` contains the test cases which test for “Circle-ness”. Both consist of re-usable test cases, and `CircleTest` automatically runs the `ShapeTest` test cases on `Circles`, to ensure that `Circles` are indeed `Shapes`.

`ShapeTest` needs to create `Shapes` before it can test them; it uses a `ShapeFactory` (passed as a reference parameter) to do this. Similarly, `CircleTest` needs to create `Circles`, for which it uses a `CircleFactory`.

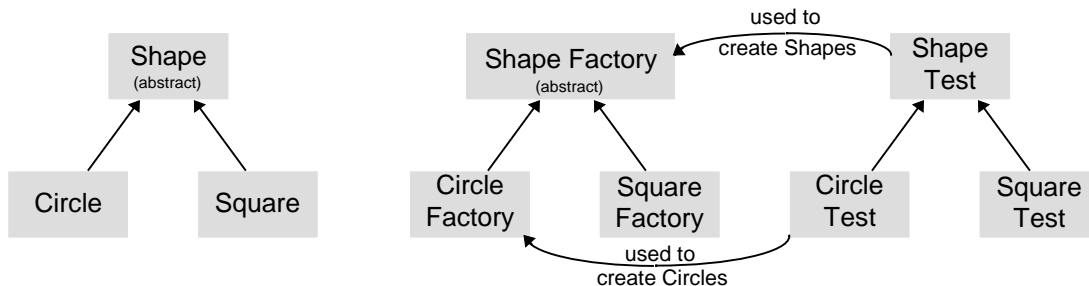
`ShapeFactory` is an abstract class, since it can’t actually create any `Shapes`. The purpose of `ShapeFactory` is to define an interface for shape-creation which the derived factory classes (like `CircleFactory`) must implement.

---

<sup>10</sup> c.f. the constrained genericity and design-by-contract ideas described in [Meyer] and supported by the Eiffel language.

CircleFactory implements the interface by creating Circles (remember that a Circle is A Shape, so this fulfils the ShapeFactory requirements). A hypothetical SquareFactory would implement the same interface by creating Squares. CircleFactory also defines an additional interface specifically for creating Circles.

When CircleTest runs the ShapeTest test cases, it passes it the CircleFactory object. This is used by the ShapeTest test cases, which test that the Circles so created are valid Shapes.



In this section we have seen how to use re-usable test cases to keep testing effort for re-usable components within reasonable bounds. But how do we ensure that a component has been tested in all necessary contexts? This question, of *coverage*, is addressed in the next section.

### 3.4 Inheritance Context Coverage

Whenever a member function is inherited, it can be re-used in a new context – acting on a Derived class object instead of a Base class object. As discussed above, we need to re-test such member functions in each new context (preferably using re-usable test cases).

To ensure that the member function has been thoroughly tested in each context, it is necessary to extend the definition of existing structural coverage metrics to take into account the *inheritance context*. For each metric, coverage is measured separately for each context. The current context is the (dynamic) type of the object on which the member function is acting.

The traditional coverage metrics can be calculated from the inheritance context coverage data by aggregating the coverage achieved across all contexts.

When calculated for a specific inheritance context, function entry-point coverage provides a basic verification that each function has been calculated in the specified context. Typical recommended coverage requirements are:

1. once-full context coverage: 100% (statement, decision) coverage in at least one context, 100% entry-point coverage in all contexts
2. strict context coverage: 100% (statement, decision) coverage in all contexts

Use of inheritance context coverage metrics can be used to ensure that re-used code has been tested in all necessary contexts.

### 3.5 State Coverage

When testing a class whose behaviour is modelled by a state machine, we will naturally use our knowledge of the possible states, and the transitions between them, to help design suitable test cases.<sup>11</sup> To ensure that we have not missed any cases, enhanced coverage metrics can be used to measure coverage separately for each possible state.

<sup>11</sup> See [Binder] for a description of one such approach.

For example, consider a `bounded_stack` class. It will have states `empty`, `normal` and `full`. To ensure that the class's behaviour in each state has been tested, coverage is measured separately for each state.

The user must provide a function which determines the "current state". In all other respects, state coverage is very similar to inheritance context coverage: coverage is measured separately for each state/context, and coverage metrics can be calculated for a specific context, or across all contexts.

Typical recommended coverage requirements are:

1. once-full state coverage: 100% (statement, decision) coverage in at least one state, 100% entry-point coverage in all states
2. strict state coverage: 100% (statement, decision) coverage in all states

Astute readers will have realised that achieving strict state coverage will often be infeasible. If the class is to behave differently in each state, then there must be some code which is executed differently, depending on the state. For example, in the `bounded_stack` class, the `push()` member function is likely to include a decision of the form `if (empty())`. Clearly, this decision will only branch true in the `empty` state, and will only branch false in the `normal` or `full` states.

A structural coverage metric for which 100% coverage is infeasible is almost useless as a measure of testedness since it is impossible to know what "75% coverage" means: "the 3 feasible cases have all been covered" or "9 of the 11 feasible cases have been covered".

To counter this problem, the enhanced coverage metrics are further modified to allow the tester to define particular coverage items (e.g. decision branches) as being infeasible in a particular state. Coverage metrics can then be calculated based on the feasible cases only: in the example above the results would be "100%" and "82%" coverage respectively – making the difference clear.<sup>12</sup>

In the inheritance context case, achieving full coverage usually is feasible. Inherited member functions should behave (mostly) the same when applied to a derived class as they do in the base class; any differences should be encapsulated as virtual member functions which are overridden. Thus, most member functions do not contain decisions based on the actual type of the underlying object. However, in those rare cases (involving the `dynamic_cast` operator) where there is a problem, the same approach can be applied.

## 4. Summary

We have seen that testing C++ software presents a number of new challenges. Increased numbers of dependencies, combined with the exposition of class implementation details in header files, make isolation testing infeasible in most cases. The resulting switch to bottom-up testing forces us to perform unit-level testing on larger and larger integrations of units. The re-use of components, through inheritance and instantiation, forces us to consider the re-use of test cases. New coverage metrics are required to maintain the value of structural coverage analysis for test case design.

This paper has presented solutions to all these problems. The use of tools implementing and supporting these solutions, such as IPL's new product *Cantata++* [IPL], will result in more efficient and more effective C++ testing.

---

<sup>12</sup> To ensure that "cheating" is not possible, the raw coverage metric is also reported.

## 5. References

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

[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

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

[Meyer] B. Meyer, *Object-Oriented Software Construction (2<sup>ed</sup>)*, Prentice Hall, ISBN 0-13-629155-4