



Practical object-oriented state-based unit testing

H.G. Reed,^a C.D. Turner,^b J.B. Aibel^a & J.T. Dalton^a

^a*Marcam Corporation, Newton, MA, USA*

^b*Microsoft Corporation, Redmond, WA, USA*

ABSTRACT

Although traditional testing technology can be applied in object-oriented software development, object-oriented software is structured differently from traditional software and thus requires new solutions. The first part of the solution tests class-based production software with class-based unit tests. The second part of the solution takes advantage of state-based testing techniques where applicable. State-based testing methodology, developed at University of Durham, England focuses on the communication of an object's operations via its state. Marcam Corporation is developing large object-oriented products (>1000 C++ classes) for the process manufacturing industry and requires a commercially viable and theoretically sound testing technology. This paper describes efforts at Marcam Corporation to produce a class-based testing technology taking advantage of state-based techniques.

INTRODUCTION

There are many competing factors determining how an organization tests its software. In the commercial environment, this includes allocating development resources, motivating developers to write unit tests, and maintaining these tests over the lifetime of the product. The central problem is focusing limited development resources on the unique aspects of unit tests, minimizing the overhead of creating test environments and packaging. Secondary factors include choosing which techniques (state-based, boundary analysis, etc.) are applicable to validate each production unit.

Marcam's use of a standard test structure solves the overhead problem. This structure is class-based, thus avoiding the paradigm mismatch of using traditional procedural approaches for testing class-based production software. The structure is generated in part by tools, reducing test drudgery and increasing motivation to produce high quality tests.

364 Software Quality Management

Marcam's development resources then focus on selecting appropriate validation techniques for each production unit. Marcam found state-based testing advantageous for validating object persistence. This technique lends itself to automated generation, thus saving more development resources.

This paper begins with a review of traditional procedural and object-oriented testing, identifying the paradigm mismatch between procedural test code and class-based production code. Next, an orthogonal architecture is introduced enabling class-based testing. The class-based test structure used to eliminate the structural overhead is described, which enables the focus to move to state-based testing, illustrated with an example drawn from the Marcam product.

TRADITIONAL PROCEDURAL UNIT TESTING

Traditional unit testing validates the function points within a software product to verify the correctness of its operation. Often, the strategy is to write a dedicated test program for each unit, requiring a harness, set up of the test environment, and producing output to describe the execution of the test, as shown in Figure 1.

Implementing this strategy is often informal and ad hoc. Frequently unit testing is abandoned because of the difficulty of this approach. When a developer determines that some code does need unit testing, he will create his own test harness, writing code to set up the test environment, run the test case, output the results and shutdown the test environment. The developer is concerned about all support issues, such as memory management, output device control and formatting, and initializing and cleaning up the application-specific environment. The actual test code is generally a small percentage of the entire unit test.

This practice also tends to be invasive, where conditional test code is added to the production code to aid in debugging. Typically this code consists of commands to output the value of variables, trace the execution of the code and perhaps validate the state of the environment. Conditionally compiled code introduces a new dimension of complexity into the production code and bugs can be introduced that only appear when this conditional code is disabled; Thielen [10] notes that this is a common problem.

One way to save time is to cut and paste code from previous test cases. The cut and paste development methodology saves time in developing the code, but it also means changes need to be propagated in multiple places in the test code, increasing maintenance cost. The amount of time which is saved by this method is entirely short term. Since the test code is typically not part of a developer's deliverables, code standards are not followed and often each developer creates their own sets of tools for managing the environment and tests. This makes it hard for developers to share unit test code outside of the development group, for example in Quality Assurance (QA) or System Integration groups. If the test code is not included in the source control system, it may become lost or just thrown away when it becomes obsolete.

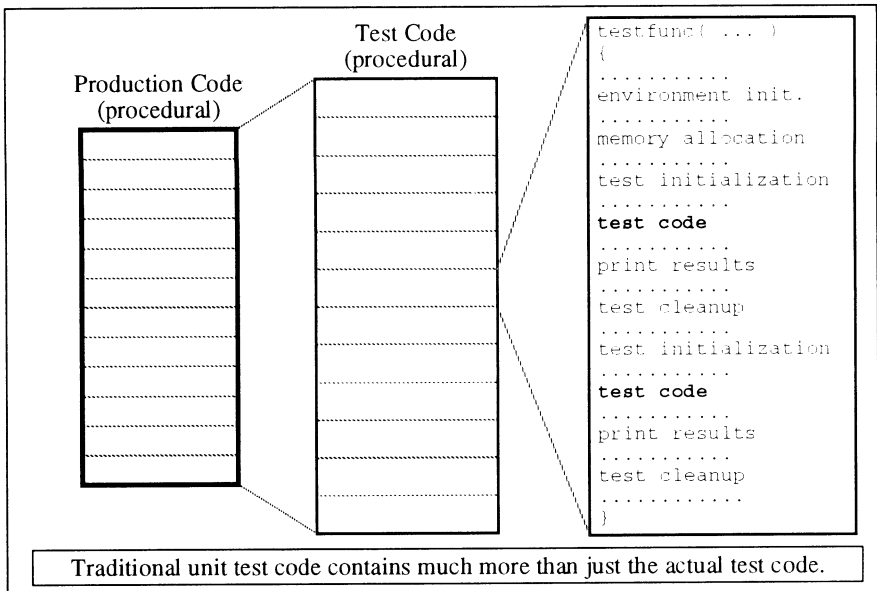


Figure 1 Traditional Unit Test Code

Some development organizations when faced with the expense of developing unit test code, or faced with the expense of maintaining an unwieldy collection of disorganized test programs halfway into a project when schedules are tight, give up on the test effort, leaving validation to the QA group. This is particularly detrimental in an object-oriented programming environment where a system test approach is less effective in detecting certain classes of errors than a unit test approach.

TRADITIONAL OBJECT-ORIENTED TESTING

Programs produced using procedural techniques are highly function oriented, whereas programs produced with object-oriented techniques have a much greater emphasis upon the data being manipulated. This places a different emphasis on the program's construction. Traditional testing techniques can be adapted to object-oriented programs, however they are not guaranteed to exercise all of the facilities that are peculiar to object-oriented programming.

Traditional practice in industry is to continue using the procedural organization of unit tests for class-based production code, as shown in Figure 2. Since the number of classes in commercial applications can be quite large, the cut and paste technique is even more costly. Most important is the paradigm mismatch between class-based production code and procedural unit testing.

Objects consist of both a state and a defined behavior (operations). The functionality of the class is defined by both the operations and the interaction between the operations and the state. Traditional testing techniques can eventually

366 Software Quality Management

exercise many of these interactions, however this is not guaranteed. Additionally, object-oriented programming uses techniques such as polymorphism which is rarely found in traditional procedural programs.

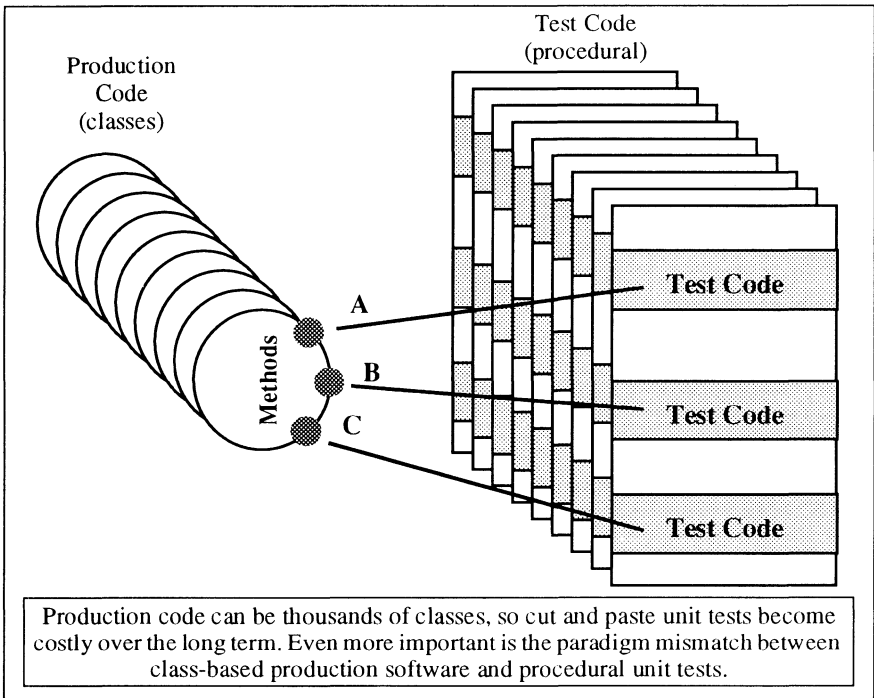


Figure 2 Traditional Object-Oriented Testing

Current research into the testing of object-oriented programs is concerned with a number of issues. The black-box nature of classes and objects implies that a black-box testing technique would be beneficial. A number of black-box approaches have been developed, including the module validation technique of Hoffman and Strooper [3], and the ASTOOT suite of tools by Frankl and Doong [2]. Both of these techniques concentrate upon the modular nature of object-oriented programs. However Fielder [1] notes that a more adequate testing technique can be gained by combining black-box testing with the use of a white-box coverage measure.

ORTHOGONAL ARCHITECTURE ENABLES OBJECT-ORIENTED TESTING

Object-oriented testing can be accomplished by extending traditional techniques, but at a greater cost. This section examines some fundamental object-oriented techniques that enable cost effective object-oriented testing.

Reed [8] discusses how fundamental object-oriented library architecture using orthogonality benefits both production and test code. Since commercial

object-oriented products are built by aggregating objects from hundreds or thousands of classes, properties can be identified that are invariant with respect to aggregation of objects. These properties make aggregation easy for development and make testing more effective. They have been proven over many commercial generations of fundamental object library work by Reed and are listed below:

- State Printing
- Memory Management
- Aggregation Mechanisms and Policies
- Test Instrumentation

State printing is a method (dump) common to all objects that outputs the internal state in a uniformly formatted manner. Dump is recursive through all levels of aggregation, and can be cleanly compiled out of a production system.

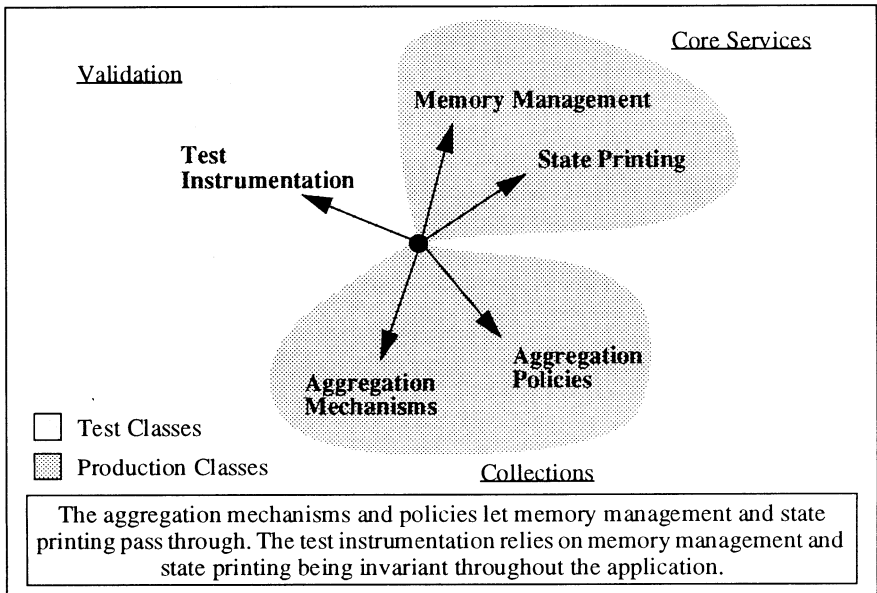


Figure 3 Orthogonal Properties Required For Effective Object-Oriented Testing

Memory management overrides the *new* and *delete* operators for all objects allocated on the heap. It also provides raw buffer allocation and deallocation. In addition, a memory manager can dump the state of all objects and raw buffers it is managing.

Aggregation mechanisms collect heap-based objects beyond what the language provides. Aggregation collects objects into a topology (linear, tree, etc.) and allows other orthogonal properties to pass through. For example, a request to print a collection causes it to print its state, then to request each object collected to print its state. Aggregation policies encapsulate decisions, and similarly allow orthogonal properties to pass through.

368 Software Quality Management

Test instrumentation is a set of classes that use the production classes, but are not used by them. These classes are examined in more detail below. Other than the dump method, there are no additional requirements placed on objects, and the instrumentation does not interfere with any of the other properties.

INSPECTION CLASSES PROVIDE AN INSTRUMENTED INTERFACE

Building on the orthogonal architecture, inspection methods package test code and inspection classes package inspection methods.

Inspection Methods

When executing a production method, it is useful to log states before and after the execution. This non-invasive instrumentation surrounding the execution of a method can be extracted and reused. Code which logs the call to the production method, entry and exit times, parameters, and return codes, can be packaged into an “inspection method.” This instrumentation uses the orthogonal state printing (dump) method, which is read-only and does not affect the state of the object.

Each of the inspection methods take the object under test as a parameter, as well as the rest of the parameters necessary for the production method, as shown in Figure 4. This packaging not only reduces cut and paste code, but also eliminates the need for specialized test code in the production class that is used only during unit testing. It is much preferred to have instrumentation outside of production code rather than compiled in, even if the compilation is conditional.

The inspection methods are capable of running against code built for general release as well as development and unit testing. With the dump method enabled during development, the unit tests provide a white-box testing environment, allowing the developer to be most effective in determining corrective actions for failed tests. Black-box test techniques are in effect when the dump method is disabled for general release. Unit tests in this form can be run as acceptance tests by QA prior to product level tests.

Inspection Classes

From the inspection methods, an “inspection class” providing a non-invasive instrumented interface is constructed. This interface includes an inspection method for each public and protected member in the production class. The inspection class interface is analogous to the production class interface. There is very little to learn in the new interface in order to construct unit tests.

The inspection class and its inspection methods is generated from a simple set of rules. A script for the pattern matching tool *awk* was developed to parse the production header files and construct inspection class headers and source files. This “inspection class generator” automates much of the rote work in constructing inspection classes, yielding savings in development time.

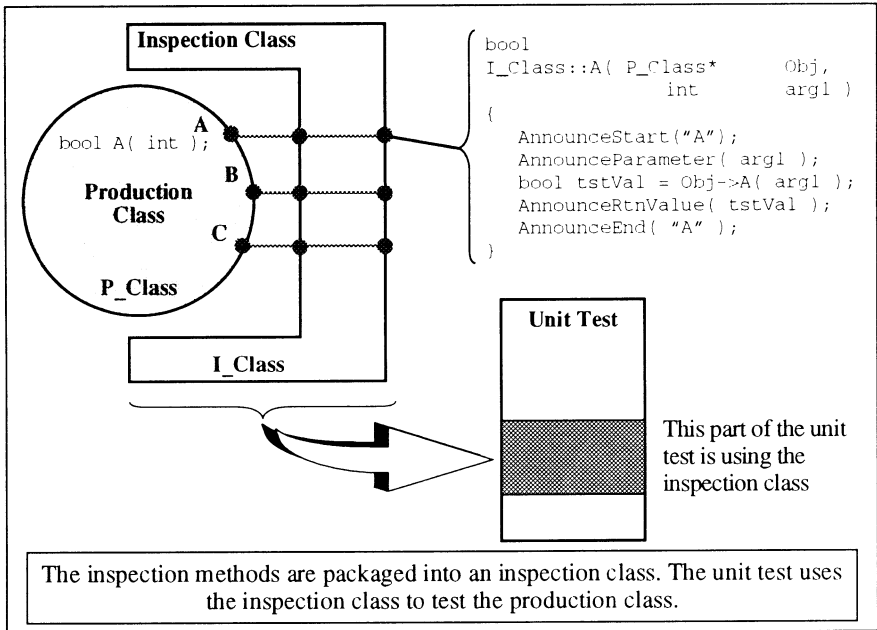


Figure 4 Package Inspection Methods in an Inspection Class

UNIT TEST SERVICES

Given a set of inspection methods grouped into an inspection class, common services of test output and test environment maintenance can be provided by test instrumentation base classes.

The kernel test class allows *infrastructure* objects to be tested in isolation. It provides services for:

- Output verbosity control
- Output formatting
- Memory management leak detection and reporting
- Expression checking
- Reporting of errors encountered

The monitor test class, derived from the kernel test class, provides additional services to application classes such as:

- Application environment initialization and cleanup.
- Test case setup and cleanup.
- Gross level coverage measurement and reporting.
- Unit test execution

The Test Subclass

The kernel and monitor classes and the inspection class do not address the

problems of accessing protected production methods or instantiating abstract classes. This is solved by a test subclass. The test subclass is derived from the production class and therefore has access to the protected methods of its parent. Inspection class access to these methods is granted by the test class declaring it a *friend*. This keeps the test technique completely non-invasive. This test subclass also provides a location for the implementation of the pure virtuals. It is expected that in most cases the pure virtuals will be implemented by empty, stubbed-out code. For those classes which do not need a test class, the test class can be replaced with an alias to the actual production class. The full inheritance tree including the inspection and test classes is shown in Figure 5.

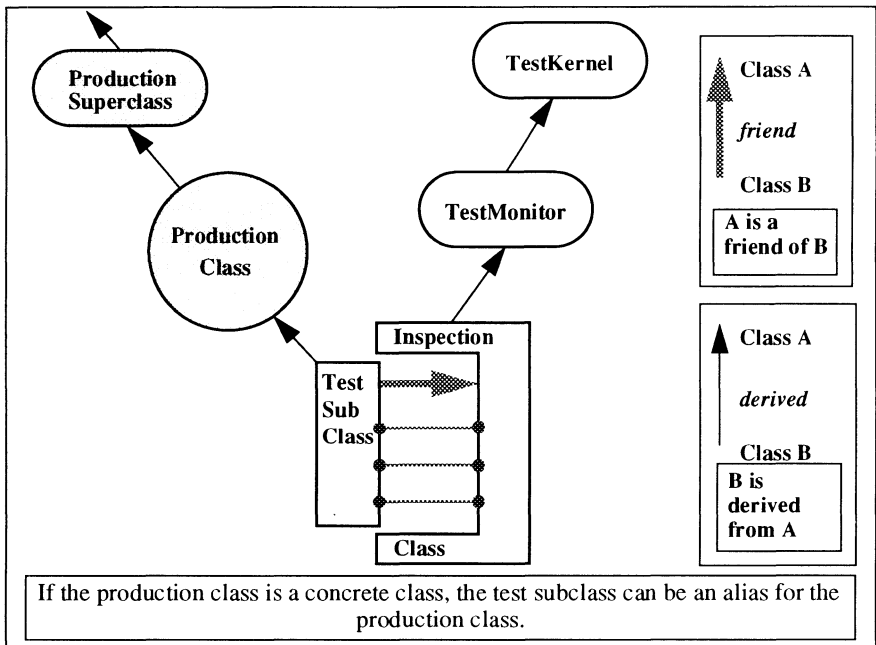


Figure 5 The Complete Test and Inspection Class Inheritance Tree

ORGANIZATION OF TEST SUITES

The non-invasive instrumented interface provided by an inspection class and test subclass can be generated automatically from a production class header file, resulting in a tremendous reduction in the amount of code the developer needs to write, allowing the developer to concentrate on designing tests. This section examines the structure of the unit tests within the package of the inspection class.

Unit tests must achieve a certain minimum standard of coverage. This is ensured by dividing the set of production class methods into groups according to behavior and testing technique. There are also dependencies between some groups and others, for example all methods depend on the constructor methods.

At Marcam, methods are tested in the following groups:

- Memory Lifecycle
- Persistence Lifecycle
- Get and Set of Attributes
- Operators
- Semantics

Memory lifecycle tests methods which affect the life of the object in memory. These include the constructors, copy constructors, destructor and assignment methods. Generally the errors which occur in the lifecycle are memory leaks and improper initialization. Lifecycle tests involve creating objects with each constructor, assigning objects to each other, then deleting them from memory and checking for leftover allocated memory.

Persistence lifecycle is for objects which are stored in a database for later retrieval or storage across invocations of the product. A state-based testing approach is used for persistent methods, discussed in detail in the next section. The states defined in the memory and persistence lifecycles are so regular that a majority of the test cases in these sections can be predefined and therefore implemented by the inspection class generator, reducing developer effort.

Get and set tests those methods which allow the access to and setting of value-based attributes. Testing assures that the values can be accessed, and that ancillary properties such as “read-only” are complied with. A state-based testing approach also works well in this group.

Operators tests the methods that perform comparison, arithmetic, logical operations and others. Operator testing is functional in nature and relies on techniques such as equivalence class partitioning and boundary value analysis.

Semantics tests behavior that makes the object unique. This is the area where the developer can focus on the complex and creative testing of the object and object interaction. Many testing techniques are employed here.

Test Suite Packaging

Individual tests are placed in a test suite by method grouping. These test suites are included in the inspection class along with the inspection methods providing the instrumented interface. Given this uniform organization for all inspection classes, it is possible to execute the set of unit test suites in a uniform manner.

Before invoking the test suite execution service on the test monitor class, an inspection object must exist. This is done by providing a standard entry point “testmain()” in the test code which is responsible for constructing an instance of the inspection class. The implementation of testmain() is a nonclass templated function which is also generated for the developer. Figure 6 shows this packaging of the inspection class.

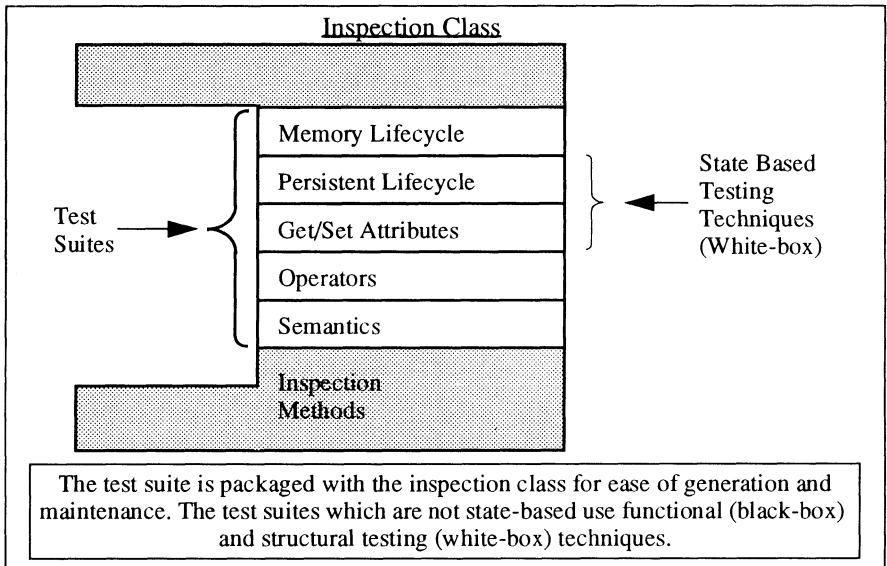


Figure 6 Test Suites are Packaged in the Inspection Class

PERSISTENCE TESTING

This section examines the application of state-based testing to the persistence of objects, which is critical to the functionality of Marcam's product. State-based testing and its applicability will be briefly described. An example of a multi-state object drawn from the Marcam project will be shown. Some additional testing problems due to the specific persistence approach will be discussed.

State-Based Testing

Of the various methods considered in developing test cases, state-based testing was the most applicable to persistence. State-based testing was developed at the University of Durham to address some of the problems in adapting testing techniques to object-oriented programming, and is fully described in Turner [11].

An object's behavior is defined by the code within its operations. Events and situations occur that have to be communicated to the other operations of the object. This communication is usually performed by storing specific values within the attributes of the class. State-based testing models these interactions as a finite-state-automata which is then used to predict the response of each operation to a particular state.

Test cases are generated to validate that at no point in an object's lifetime can it be placed into an undefined or incorrect state. All operations are validated for the state changes which they are defined for, operations can also be validated for their robustness when faced with an illegal state situation.

Example

An example, drawn from the Marcam project, deals with persistent classes. Persistence is implemented by a root persistent class from which all persistent objects are derived. The persistence paradigm requires six states that an object can enter:

- New - the object was created and exists in memory only
- Saved - the object exists in the database and not in memory
- Open for Read/Write - the object exists (and is identical) in memory and the database and the client may save changes to the object
- Modified Read/Write - the object exists in memory and the database with different values and the client may make changes to the object
- Open for Read Only - the object exists (and is identical) in memory and the database and the client may not make changes to the object
- Deleted - the object does not exist in memory or the database

A state diagram, showing some of the legal transitions between these states is shown in Figure 7.

Additional Testing Problems

Since the root persistent class is inherited by all persistent classes, these classes share the above states, although the implementation is made specific to each class by providing some of the persistence code in a code template that is instantiated for each class. The test cases for these objects should also be similar, with minor changes from class to class. Because the persistence functions are critical to the operation of the Marcam product, it is not sufficient to test the functions solely at the root level. The critical nature of these functions and the templated implementation leads to the decision to test the operation of the persistence functions for each persistent class.

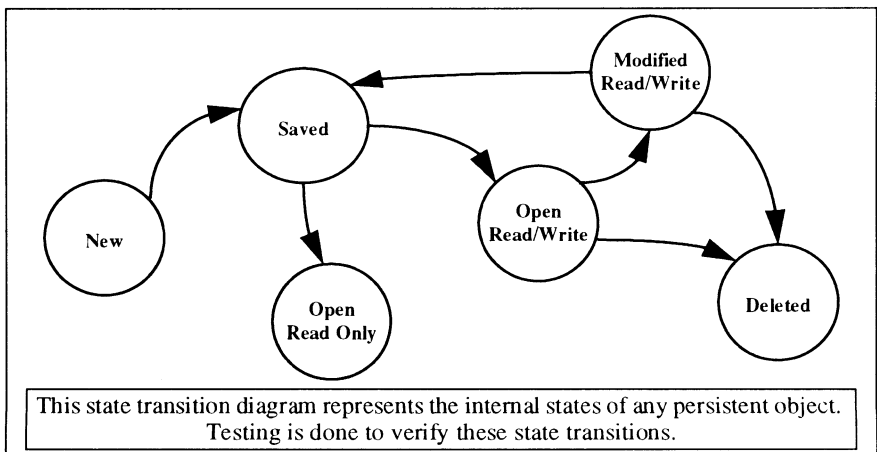


Figure 7 State Transition Diagram for Object Persistence



374 Software Quality Management

The situation of shared critical functionality leads to another testing problem. Given the similarity of the tests, interest in rewriting the same test cases over and over for the persistent classes is likely to be low and without the cooperation of the developers, the policy is likely to fail. A standard set of persistence test cases, based on the state analysis, is implemented in a code template. The inspection class generator notes the inheritance from the root persistent class and then includes the code template into the persistence test suite, as seen in Figure 8.

Benefits of This Approach

This solution allows each class to be tested with a minimum set of test cases, assuring a certain level of test coverage. The developer must supply a test setup routine specific to the class under test. This is a short and well-defined piece of code, and the inspection class generator supplies a function body skeleton. Then the developer can code additional test cases as deemed necessary to the particulars of the class.

An additional advantage to this approach is that it provides a single point for the maintenance of the test cases. If the persistence API should change, the amount of test code to be changed is minimized.

This does not appear to be an unique situation; it seems likely that there will be other cases where a whole model - an API and series of states - is shared among many classes. Using state-based testing provides a significant savings here by allowing the development of a template of standard cases which are shared by the inspection classes. It should be noted that this approach not only demonstrates the correctness of the code under test but also demonstrates the correct integration of the shared production code template.

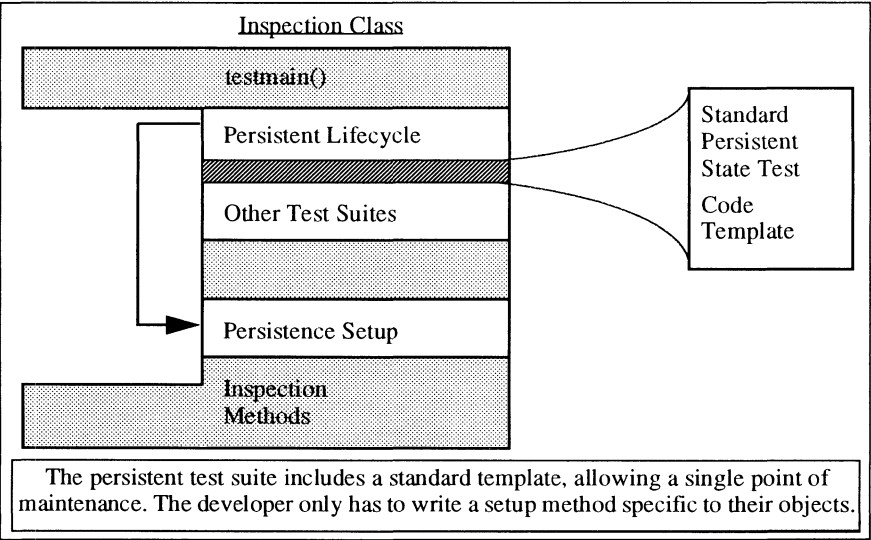


Figure 8 Inclusion of Standard Test Scenarios in the Inspection Class



RUNNING THE TESTS

The standard construction of the test suite allows for the use of a single test harness to invoke the test suite. This enables unit tests to be run by engineers unfamiliar with the production code, such as QA and System Integration groups.

UNIT TESTING DEPENDENCIES

Unit testing starts with the simple independent classes and then moves onto application classes. This tests the application as if it were an operating system; as lower levels are built and tested, those objects become trusted, and can be used to validate objects which are not yet trusted.

Co-dependent Objects

This approach works well for simple dependencies, but it breaks down for co-dependent objects. These objects are dependent upon each other, so a call dependency graph contains cycles. In an aggregate these co-dependencies may stretch several levels of a containment hierarchy as shown in Figure 9. Often co-dependent objects can't be initialized except in the context of the complex aggregate.

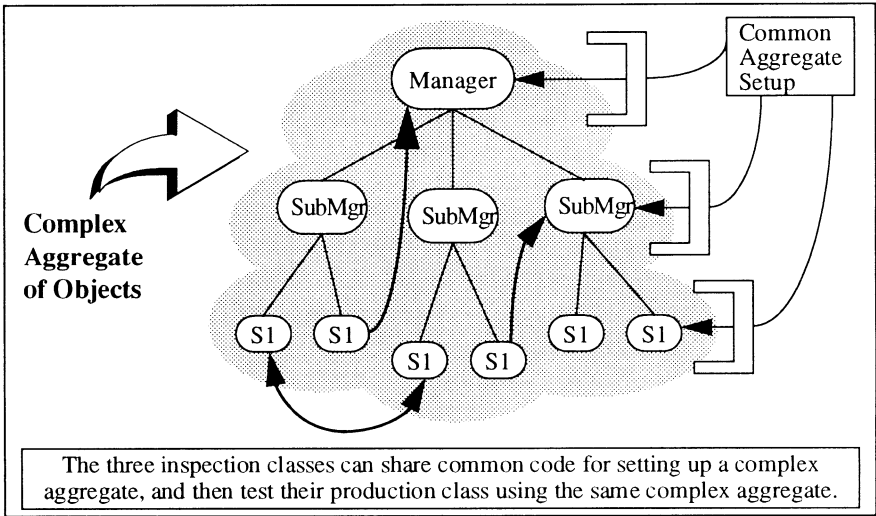


Figure 9 Strategy for Unit Testing Co-dependent Objects

The strategy here is to test each object in the context of the entire aggregate. This results in overlapping coverage, but does guarantee test coverage. When possible, the aggregate setup is made common across all the inspection classes for the aggregate. The methods which rely on the co-dependencies are implemented in the dependent object's inspection class. Thus if the SubMgr and the S1 objects depend on each other, then methods which rely on the SubMgr→S1 dependency

376 Software Quality Management

are tested in the SubMgr inspection class and methods which rely on the S1→SubMgr dependency are tested in the S1 inspection class.

MOTIVATING UNIT TESTING

One of the classic problems of unit testing is maintaining the developer's interest in writing the unit tests long enough to get a complete set of tests. Writing tests is seen as a dull, repetitive task. In the authors' experiences, there are some engineers who understand the necessity of unit testing, but a majority of engineers seem to feel that unit testing keeps them from their next engineering task.

This problem is exacerbated by the fact that the most basic steps in developing test cases, Boundary Value Analysis and Equivalence Class Partitioning, are simple and repetitive activities. There is no "engineering" involved in developing test cases with these methods. There are already several collections of common faults, such as Johnson [5] and Spohrer [9]. Marick [6] is a catalog of test conditions, making this aspect of unit testing as simple and mindless as a table look-up. And yet no test suite would be complete without these test cases.

An advantage to the unit test strategy outlined in this paper is that it minimizes the code the engineer has to write to implement test cases like these. There is no need for the developer to spend any time developing test overhead code, except for set-up code specific to their module. This frees the engineer to concentrate on designing unit tests, rather than the rote work.

The focus on state-based testing re-enforces this effect. Not all testing is rote-work; there is a spectrum of testing methods along a creativity scale, as shown in Figure 10. State-based testing, path testing and even error guessing are high in the scale and engage a developer, requiring creativity and engineering skill. Using these methods makes the test development more interesting and motivates a developer into implementing higher quality unit tests.

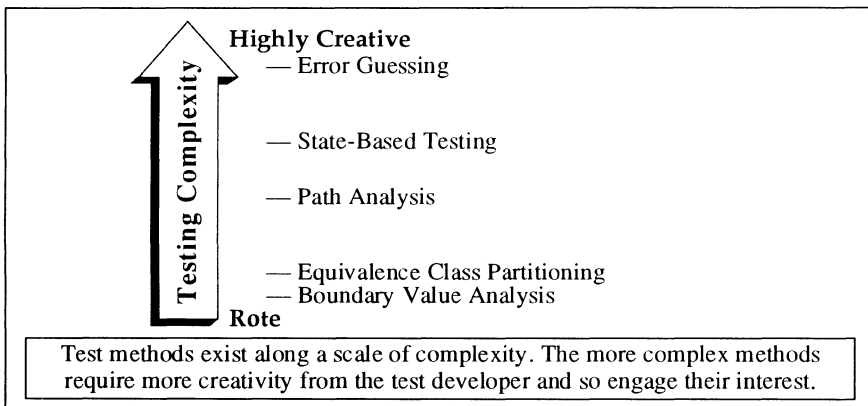


Figure 10 Testing Methods in Order of Complexity.

In addition, the uniform look and feel of unit tests in this approach reduces the burden of routine details, and encourages the design of unit tests that will have a long lifetime. The inspection class generator offers options to generate the simple rote tests that can be deterministically described. For example, the memory lifecycle cases that test constructor-destructor calls for memory leaks are created by the inspection class generator.

Lastly, the standardized nature of the unit tests makes reviewing the test design and code easier, allowing improved management of unit test coverage.

STATUS

Marcam has implemented this unit test strategy. Starting with simple independent objects, Marcam is now testing persistent and aggregate objects. The next step, underway as of this writing, is expanding the strategy for co-dependent objects. The authors expect the support required for co-dependent objects will be similar to the work described in the persistence testing section.

CONCLUSION

The reality of commercial software is the limited availability of development resources allocated to testing. The first step is eliminating structural overhead and the second step is selecting appropriate test techniques. Yet no test technique is a panacea without addressing the practical issues. The biggest cultural challenge in implementing this strategy has been training Marcam's development community.

This strategy has increased the unit test coverage at Marcam, and made maintenance more cost effective. The authors feel that this approach has allowed Marcam to increase the quality of its product with limited development resources.

FUTURE WORK

Marcam will continue work in the following areas: inspection class hierarchy, exception testing, performance testing, and large scale scenario testing.

The inspection class hierarchy can be enhanced by making it analogous to the production hierarchy. This can be accomplished with small additional effort in the inspection class generator. This makes the testing of virtual methods easier, since they can be tested at each level of the inheritance hierarchy.

Current exception handling terminates the application. This makes testing exceptions extremely tedious. In the future, test cases that throw an exception will record this result, and continue with the remaining unit tests.

Performance testing can take advantage of the test structure by adding instrumentation for time measurement.

Large scale test scenarios will allow multiple families of complex aggregate objects to be constructed for product performance and stress testing. An ideal



378 Software Quality Management

design will allow runtime configuration of the test scenario. This will provide the flexibility necessary to simulate many different customer profiles.

Finally, metrics will be defined and collected to measure improvements in software quality due to improved unit testing.

REFERENCES

- [1] Fiedler, S.P., Object-Oriented Unit Testing, *Hewlett-Packard Journal*, pp. 69 - 74, April 1989.
- [2] Frankl, P.G. and Doong, R., Tools for Testing Object-Oriented Programs, in *Proceedings of the 8th Pacific Northwest Conference on Software Quality*, pp. 309 - 324, 1990.
- [3] Hoffman, D., and Strooper, P. Graph-Based Module Testing, in *Proceedings of the 16th Australian Computer Science Conference*, pp. 479 - 487. Australian Computer Science Communications, Queensland University of Technology.
- [4] Hoffman, D., and Strooper, P. Graph-Based Class Testing, in *Proceedings of the 7th Australian Software Engineering Conference (ASWEC)*. 1993.
- [5] Johnson, W.L., Soloway, E., Cutler, B., and Draper, S. W. Bug Catalogue: I. Yale University Technical Report, October, 1983.
- [6] Marick, Brian. A Test Condition Catalog. Testing Foundations, Champaign IL, 1992
- [7] Marick, Brian. Three Ways to Improve your Testing, in *Proceedings of the 10th Pacific Northwest Conference on Software Quality*, 1992.
- [8] Reed, H.G., Object-Oriented Design by Orthogonality, *Computer Language*, vol. 9, no. 1, pp 51 - 56, January 1992.
- [9] Spohrer, J. C., Pope, E., Libman, M., Scak, W., Freiman, S., Littman, D., Johnson, L., and Soloway, E. Bug Catalog: II, III, IV. Yale University Technical Report YALEU/CSD/RR#386, May 1985.
- [10] Thielen, David, *No Bugs! Delivering Error-Free Code in C and C++*, Addison-Wesley Publishing Co., 1992.
- [11] Turner, C.D. and Robson, D.J., The Testing of Object-Oriented Programs, Technical Report TR-13/92, University of Durham, England, 1992.