JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2005

Vol. 4, No. 5, July–August 2005

A Study of Test Coverage Adequacy in the Presence of Stubs

Errol L. Lloyd, Computer and Information Sciences, University of Delaware, Newark, DE 19716, USA
Brian A. Malloy, Department of Computer Science, Clemson University, Clemson, SC 29643, USA

The purpose of implementation-based testing is to gain a measure of confidence in the correctness of the software by providing adequate coverage of the code. One unit of testing in object-oriented software is a class. However, classes use other classes and if class interactions form a cycle of dependencies then, to test a client class that uses an untested supplier class, stubs must be constructed to simulate the correct behavior of the untested supplier class. However, an important consideration in using stubs to facilitate testing of a client class is the adequacy of code coverage in the client when the client uses a stub rather than the actual supplier class.
In this paper we present a study of test coverage adequacy in the presence of stubs as used to facilitate testing of an object-oriented application. We examine coverage in both cluster and class-based testing of the application and, using the results of our study, we draw some conclusions about the coverage provided when stubs are used in each form of testing. In addition, we study the ability of the tester to meet the specified adequacy criteria for class-based testing in the presence of stubs.

1 INTRODUCTION

The purpose of *implementation-based testing* is to gain a measure of confidence in the correctness of the software by providing adequate coverage of the code. Here, a program is considered to be *adequately tested* if it has been covered according to the specific test criteria: for example, whether all statements, branches or data flow paths have been executed. However, in testing object-oriented software, there can be no confidence in the correctness of the program if (reachable) parts of the program have never been exercised [21].

When testing object-oriented software, the basic unit of testing is the class and there are several strategies referenced in the literature for testing classes [1, 6, 7, 8, 9, 11, 12]. The difficulty in testing classes is that classes interact with other classes. If class A uses class B, then A is said to be a client of B and B is a supplier of class A. One of the challenges in such testing is to order the classes for testing in the presence of class interactions or dependencies [5, 14, 15, 17]. To order the classes, an object relation diagram, ORD, is constructed where the nodes in the ORD are classes and the edges express dependencies between the classes. If there are no cycles in the ORD then a reverse topological order of the nodes in an ORD will produce a

Cite this article as follows: Errol L. Lloyd, Brian A. Malloy: "A Study of Test Coverage Adequacy in the Presence of Stubs", in *Journal of Object Technology*, vol. 4, no. 5, July–August 2005, pp. 117–137,
http://www.jot.fm/issues/issues_2005_07/article3

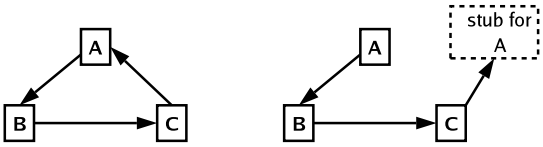


Figure 1: *Using a stub.* The left side of this figure shows an ORD with a cycle of dependencies. The right side of the figure illustrates elimination of the cycle by removing the edge from node C to A, with a stub used in place of untested class A.

class order for testing that may obviate the propagation of errors.

However, if an ORD contains a cycle of dependencies, such as the one on the left side of Figure 1, there is, in general, no ordering of the nodes that is guaranteed to prevent the propagation of errors when testing a client class that uses an untested supplier class. For an ORD with cycles, previous research has proposed the removal of edges in the ORD until cycles are broken and, to test a client class that uses an untested supplier class, *stubs* should be constructed to approximate the behavior of the untested supplier class [14, 15, 17, 24]. For example, the ORD on the right side of Figure 1 illustrates the removal of the edge from C to A, with a stub approximating the behavior of class A. To test these classes, first C is tested using a stub for A, then B is tested using the tested C and the stub for A. Finally class A is tested using B and C, with C using A rather than a stub for A.

However, an important consideration in using stubs to facilitate testing of a client class that uses an untested supplier class is the adequacy of code coverage in the client when the client uses a stub rather than the actual supplier class. If the client uses a stub that provides only the minimum functionality of a supplier class, as proposed in references [3, 5, 14, 16, 25], than the suite of test cases may not provide the same coverage in the client that was provided when the actual supplier was used. As a result, the client may not be adequately tested according to the coverage criteria established for the application or for the particular section of the application under test. Moreover, if an error is uncovered when testing a third class that uses an inadequately tested client class, a tester may not safely assume that the third class contains the uncovered error. More importantly, an application may be released into production under the assumption that it has been adequately tested when, in fact, parts of the program may remain unexercised. The negative impact of inadequate testing is well documented [20, 28].

In this paper we investigate test coverage adequacy in the presence of stubs as used in testing classes or class clusters in an object-oriented application. We begin in the next section by rigorously defining the concept of a minimal stub, and by providing a classification of the kinds of methods that can occur in an object-oriented application. We further describe the construction of a minimal stub for each kind of method in our classification. In Section 3 we discuss specifics related to the adequacy of both cluster testing and class-based testing as concerns the use of stubs. The relevant features of the application used in our case study are described

in Section 4. Sections 5 and 6 are devoted to presenting the results for our case study application on test coverage adequacy in the presence of stubs. We describe results for both cluster and class-based testing and draw some important conclusions about the coverage provided when stubs are used in each form of testing. We conclude in Section 7.

2 A MINIMAL STUB

Binder defines a (server) *stub*, as a temporary, minimal implementation of a method to increase controllability and observability during testing [3]. He continues that stubs are necessary to break cyclic dependencies, to implement deferred components in top-down integration and to test exception handling. Most references recommend a minimal implementation of a stub to eliminate the introduction or propagation of error and to reduce the time for testing [3, 5, 14, 16, 25]. The specifics of what constitutes a minimal implementation have typically been left unspecified.

In the next subsection we specifically define the concept of a minimal stub, and in the following subsection present our classification of methods and a corresponding stub. Our classification captures all methods in our case study application and is somewhat specific to C++; however, the classification is adaptable to other object-oriented languages. In the final section we provide an illustrative example of a class stub.

Minimal Stub Criteria

We measure coverage when stubs are used in cluster or class-based testing (both are described in Section 3) of object-oriented applications and, in particular, a C++ application [17]. In that context, we adopt the definition of a *stub* as a sequence of code that approximates the behavior of a class or method and that can replace the class or method instance in a system for testing or prototyping [3]. We use two kinds of stubs: a *class stub* that approximates the actions of a class and a *method stub* that approximates the actions of a member function of a class. In all of the experiments that we conduct we build a stub for a class, and then build *method stubs* for all of the methods of the class.

We propose the following criteria to specify a minimal class or method stub:

1. A minimal stub must enable the program to run to completion for cluster or class-based testing.
2. A minimal stub may not call other methods.
3. A minimal stub must consist of straight-line code, devoid of decision or loop statements. The only exception is an *iterator stub* (defined below), which may contain an **if/else** statement with a simple condition.

The above definition of a minimal stub is based on the premise that the larger the stub, then (1) the larger the possibility that the stub itself will contain an error; (2) the longer it will take to write the stub; and, (3) the longer the testing process will take, since a nontrivial time will be required to execute the stub.

A Classification of Methods

We propose the following classification of methods, and therefore the kinds of stubs that can occur in an application: *observer*, *accessor*, *factory*, *mutator*, *iterator*, *constructor*, *destructor*, and *pure virtual*. We define each type of method/stub as follows:

- *observer method* – A method that does not modify the state of the object and does not return a value. An observer method may print output to a screen or file or may return to the client with no action. An *observer stub* will use straight-line code to model the observation or it may have an empty body.
- *accessor method* – A method that returns a primitive type, possibly first computing the value that is returned. The corresponding *accessor stub* will return a reasonable value within an acceptable range.
- *factory method* – A method that builds and/or returns an object. The corresponding *factory stub* must use the default constructor to build the object.
- *mutator method* – A method that modifies the state of the object. The corresponding *mutator stub* must approximate the behavior of the original method using straight-line code.
- *iterator method* – A method that iterates through or returns a value in a container. One problem with an iterator method is that if the corresponding stub always returns the same value, the client might loop forever. Thus, an *iterator stub* should first return an object whose type corresponds to the type of object in the original container and then, on subsequent calls, return additional values in the same fashion, until finally returning a value indicating the end of iteration through the container.
- *constructor method* – A default, conversion or copy constructor of a class. The *constructor stub* for a default or conversion constructor will initialize the data attributes of the class to reasonable values. The *constructor stub* for a copy constructor will use straight-line code to make a shallow copy of the object.
- *destructor method* – A destructor of a class. The *destructor stub* must approximate destruction of the corresponding object using straight-line code.
- *pure virtual method* – a pure virtual method causes the containing class to be abstract and usually does not include an implementation. We exploit the

```
( 1) enum EdgeTypes { association = 0, inheritance,
( 2)       composition, dependence, polymorphic,
( 3)       ownedElement, merged, aggregation, none };
( 4) class EdgeType {
( 5)  EdgeTypes thisEdge;
( 6) public:
( 7)  EdgeType() : thisEdge(association) {}
( 8)  EdgeType(EdgeTypes l) : thisEdge(l) {}
( 9)  EdgeTypes getType() const { return thisEdge; }
(10)  void setType(EdgeTypes t) { thisEdge = t; }
(11)  string getString() const {
(12)    return convertToString(thisEdge);
(13)  }
(14)  static string convertToString(EdgeTypes t) {
(15)    if ( t == association ) return "association";
(16)    else if ( t == inheritance ) return "inheritance";
(17)    else if ( t == composition ) return "composition";
(18)    else if ( t == dependence ) return "dependence";
(19)    else if ( t == polymorphic ) return "polymorphic";
(20)    else if (t ==ownedElement) return "ownedElement";
(21)    else if ( t == merged ) return "merged";
(22)    else if ( t == aggregation ) return "aggregation";
(23)    else return "none";
(24)  }
(25)  static EdgeTypes
(26)  EdgeType::convertToEnum(const string&);
(27) };
```

Figure 2: *C++ code for the original class* EdgeType.

direct method for stubbing a pure virtual method [18] and provide more detail in Section 5.

The above classification covers all of the methods in our case study application. Of course some of the methods in our C++ application are actually a combination of types of methods. For example, a method that overloads assignment is both an *accessor*, usually returning a reference to the object, and *mutator*, assigning new values to the object. We require that method stubs have the same name, signature and return type as the original method. Also, we do not collapse overloaded methods into a single method since some developers use overloaded methods to choose among implementation alternatives.

Sample Stub Class

Figure 2 depicts a class, EdgeType, that is part of a larger system (COS) that implements a particular variety of ORD and is fully described in Section 4. Lines 1–3 of Figure 2 list an enumerated type, EdgeTypes, which represents the types of edges in an ORD. The class listed on lines 4–27, EdgeType, is an abstraction of the types of

```
( 1) class EdgeType {
( 2)  EdgeTypes thisEdge;
( 3) public:
( 4)  EdgeType() : thisEdge(association) {}
( 5)  EdgeType(EdgeTypes l) : thisEdge(l) {}
( 6)  EdgeTypes getType() const { return thisEdge; }
( 7)  void setType(EdgeTypes t) { thisEdge = t; }
( 8)  string getString() const {
( 9)    return "association";
(10)  }
(11)  static string convertToString(EdgeTypes t) {
(12)    return "association";
(13)  }
(14)  static EdgeTypes
(15)  EdgeType::convertToEnum(const string&);
(16) };
```

Figure 3: *C++ code for a stub class for* EdgeType.

edges in an ORD, with methods to convert an edge to/from an enumerated type or string including method convertToString, lines 14–24, and method convertToEnum, lines 25–26 (the body of convertToEnum is elided for brevity).

A corresponding stub for the class in Figure 2 is shown in Figure 3. The default constructor of class EdgeType, line 7, remains the same in the stub, line 4. However, the conversion constructor of class EdgeType, line 8, has been simplified in the stub, line 5; our classification of stubs demands only that we initialize data attributes to reasonable values, and we here choose a simple initialization. A larger simplification is illustrated for the observer method, convertToString, where in the class, lines 14–24, an **if/else if** statement chooses from among nine alternatives; however, in the stub, lines 11–13, a reasonable value is returned, association. For class EdgeType in Figure 2, lines 7 and 8 list *constructor methods*, line 9 lists an *accessor method*, line 10 lists a *mutator method* and lines 11–13 list a *factory method*.

3 TWO TESTING METHODOLOGIES

A wide range of approaches to testing, based on varying test granularities, have been proposed [1, 3, 6, 7, 8, 9, 19]. In this paper we study the effect of stubbing relative to two of these approaches: *cluster testing* [3, 18, 19] and *class-based testing* [1, 6, 7, 8, 9, 11, 12]. In the next two subsections we elaborate on the experimental methodologies that we utilize in conjunction with these two approaches.

For both types of testing, we focus exclusively on *statement coverage*, where the testing goal is to execute as many (hopefully all) of the statements as possible. To measure statement coverage, we use *gcov*, a GNU tool included with the GNU *gcc* compiler suite. *gcov* is a test coverage program that can provide profile information and execution frequencies for statement and branch coverage in a program [2].

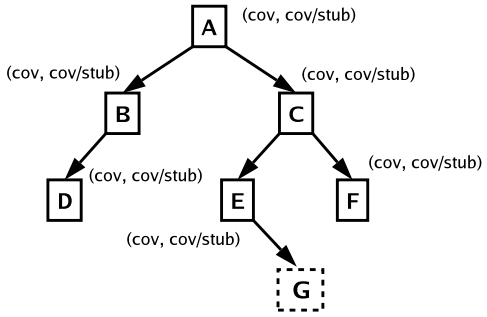


Figure 4: *Coverage in the presence of a stub for G.*

Coverage in Cluster Testing

The objective in *cluster testing* is to test a set of related and interacting classes as a unit. In many instances, this may encompass an entire object oriented program, in which case the test cases are actual program inputs.

Our investigation of coverage provided by cluster testing in the presence of stubs entails the use of: (1) an ORD to represent the classes and the dependencies among the classes in the application under test, (2) cluster level test cases, and (3) a stub for each of the classes in the ORD.

In this approach, we first execute each of the test cases in the test suite using the original classes in the application, using gcov to measure the statement coverage for each of the classes. Then, using a reverse topological ordering of the nodes in the ORD, we choose a class in the ORD, replace that class with a stub, and measure the coverage for each of the remaining classes in the ORD. We then replace the stub with the original class and choose another class in the ORD to be replaced with a stub and measure the coverage for the remaining classes in the ORD. We continue this process until each of the classes, except for root classes in the ORD, have been stubbed.

To illustrate our approach to measuring coverage in testing a *class cluster*, consider the ORD shown in Figure 4, with root node A, leaf nodes D, G and F, and interior nodes B, C and E. We first measure the statement coverage provided by the test cases in our test suite when executed on the original system. We then choose a node, say G, to replace with a stub and, using the same test cases, we measure the coverage in classes A through F in the ORD. Figure 4 provides a pictorial illustration of this first stub replacement: class G is in italics to indicate that we are using a stub for G, and each of the remaining classes is annotated with an ordered pair $(cov, cov/stub)$ where the first item in the pair indicates the percentage of statements that are covered in the respective class using the original class G, and the second item in the pair indicates the percentage of statements that are covered in the respective class using the stub for class G. For example, for the ordered pair attached to class C, the first item in the pair is the percentage of statements that

are covered in C using the original class G, and the second item is the percentage of statements that are covered in C using a stub for G. We then replace the stub for G with the original class, and choose another node in the ORD whose corresponding class will be stubbed. We continue this procedure until we have successively stubbed nodes B through G in the ORD.

Coverage in Class-based Testing

The objective in *class-based testing* is to individually test each class of a program. By testing at this finer level of granularity, the execution of the class methods can be more tightly controlled, and 100% coverage is generally possible. The difficulty with class-based testing is that it is usually not possible to simply exercise a method of a class without prior execution of other code. For example, if the method is to return the number of edges in a graph, a graph must exist for that method to operate on. The key question then is to locate a valid sequence of method calls that allow the execution of a given method of a given class.

In this context, we use *contracts* as specified with the Object Constraint Language, OCL [4], to facilitate the generation of sequences of method calls for the particular class under test.

Method Sequences

A *Method Sequence Graph*, MSG, is used to generate *method sequences*, which are sequences of method invocations to test a class [13]. An *MSG*, G(N, E) for a class consists of a set of nodes, N, with one node for each method in the class. To obtain E, the set of edges in G, we first construct a complete graph on G so that each pair of distinct nodes in G is connected by an edge. Note that there are no self-loops[1]. Then, using a contract to express constraints on method pre- and post-conditions, we remove any edge between two methods O_1 and O_2 where the post-condition of O_1 violates the pre-condition of O_2. The set E consists of the edges that remain after such invalid method invocation edges are removed. Each path in an MSG corresponds to a possible message sequence, though whether or not that path is actually a message sequence depends on the full path. It is incumbent upon the test designer to insure that each pre- and post-condition is in fact satisfied for the methods on each selected path.

Thus, for our experiments we use the MSG as a guide in hand selecting a set of message sequences together with the corresponding test suite (as required) for each method in that sequence, so that in their totality, this set of method sequences exercises every statement in every method of the class. Note that in this set, the message sequences need not be unique, as they may also vary in regard to the

[1]The focus of this paper is code coverage. If our focus were testing we would use self loops, since a call to the same function twice in a row can expose security holes [28].

corresponding test suite. Indeed, it may require several method sequences to exercise every statement in a given method, due to the presence of exceptions, branching and other control flow statements.

Our investigation of coverage provided by class-based testing in the presence of stubs entails the use of: (1) an ORD to represent the classes and the dependencies among the classes in the application under test, (2) a stub for each class, (3) a contract for each class, (4) a method sequence graph and method sequences for each class, (5) a test suite for each class, and (6) a driver for each class under test.

For the ORD and stubs in items (1) and (2), we reuse the ORD and stubs employed in cluster testing. For item (3) we manually write pre- and post-conditions for each method in a class, and invariants on the data attributes of the class. We use the OCL to express the contracts. For item (4) we generate the MSG, and then for each class, generate method sequences as described above. For items (5) and (6), we write a driver and test suite so as to provide coverage (i.e. an execution) of each statement in each method of the class under test.

4 COS - A CASE STUDY APPLICATION

The results in Sections 5 and 6 form a case study of one object oriented application program, the *Class Ordering System*, COS, of Malloy, et al. [17]. This is a flexible framework for generating an integration order of classes for class-based testing. This method is driven by a cost model that assigns weights to the edges of an ORD (see Section 1 and Figure 1). The tunable parameters assign weights to the edge types in the ORD and provide flexibility in guiding the edge removal process.

Figure 5 contains a UML class diagram of a class cluster that contains the important classes and relationships in the COS; some of the relationships among the classes have been elided from the diagram for clarity. The three classes at the top of the figure, Graph, Node and Cluster, are used to build an ORD. The class at the middle left of the figure, GraphManager, is a version of the Singleton design pattern [10] and choreographs the actions of the COS. The inheritance hierarchy at the bottom of the figure consisting of Command and it's five derived classes is a version of the Command pattern [10], encapsulating the functionality for building an ORD, counting edges, finding cycles, and breaking cycles in an ORD. The class that summarizes the cost model, CostModel, is shown in the lower right of the figure. Both the CostModel class and Graph hierarchy use class EdgeTypes, shown in the middle right of Figure 5; EdgeTypes encapsulates the types of edges in an ORD.

COS was selected as the basis for the case study because it is a moderately large object-oriented program containing a range of classes and methods. The ORD for this application contains cycles, although that is not critical to the case study performed here, where the goal is to study the effect of stubbing on the adequacy of test coverage.

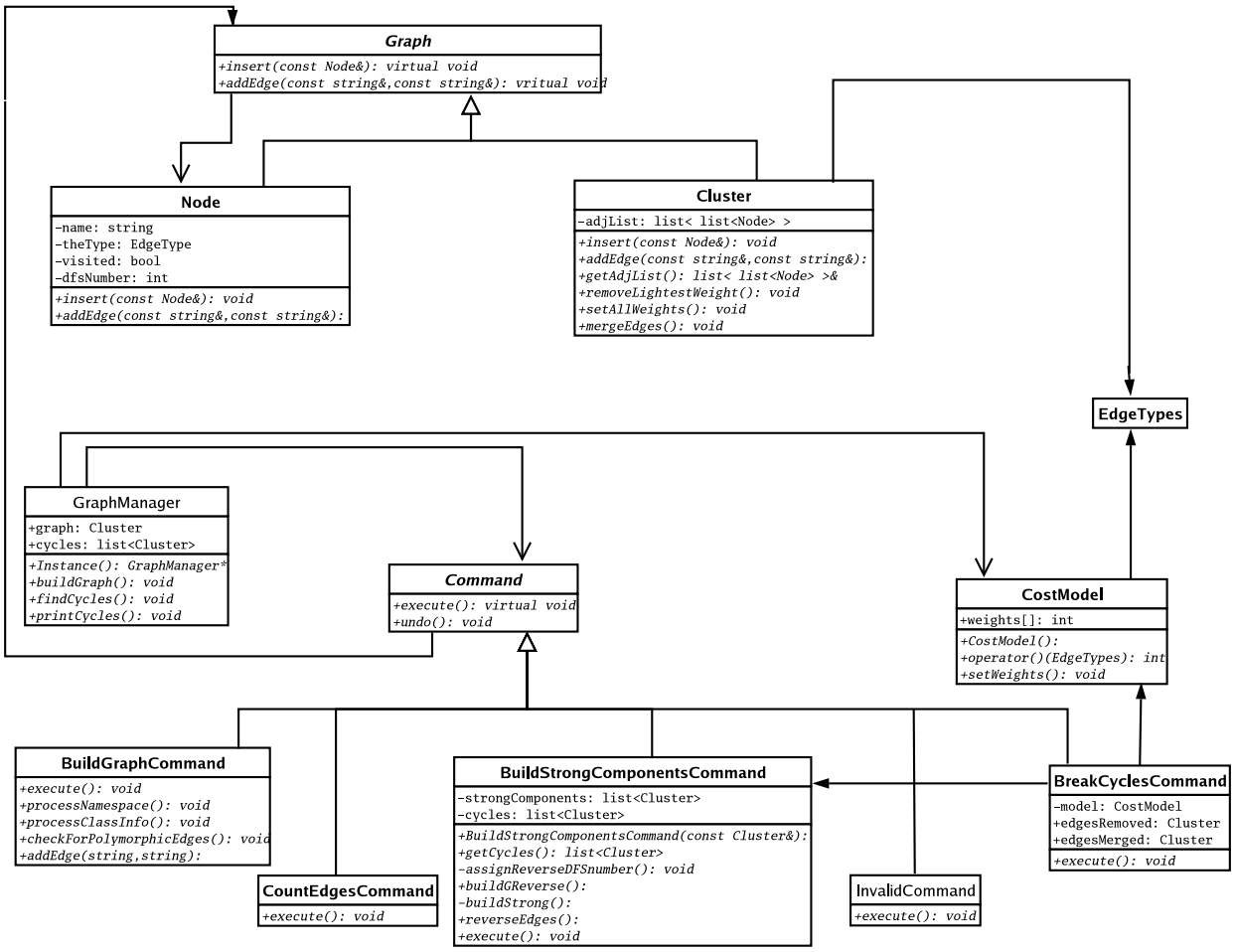


Figure 5: *Class Ordering System.* This figure illustrates a UML class diagram summarizing COS.

5 RESULTS FOR CLUSTER TESTING

In this section, we provide results from our study of coverage adequacy in the presence of stubs when they are used to test a cluster of classes. The application used in this case study is the COS as described in the prior section. All of our experiments are executed on a $DellPrecision^{TM}$ 530 workstation with $Intel©\ Xeon^{TM}$ 1.7 GHz processor equipped with 512 MB of RDRAM, running the Red Hat Linux 9.0 operating system. The COS consists of 38 classes and 1,304 lines of C++ code [23], compiled with GNU *gcc* version 3.2. We use GNU *gcov* to measure statement coverage of the COS cluster.

Our experiments use a suite of seven test cases: Adol-C, Class Ordering System (COS), ep matrix, vkey, IV Edraw, IV Graphdraw and IV Drawserv. In our study, a *test case* is a program that we use as input to the system or class. The test cases were chosen for their range and variety of application. Specifically, test case Adol-C is a

Class name	LOC in original	LOC in stub	Min
InvalidCommand	5	5	yes
Graph	9	9	yes
Command	11	11	yes
EdgeType	27	16	yes
BreakCyclesCommand	61	20	no
CostModel	68	35	yes
CountEdgesCommand	82	29	yes
Node	132	121	yes
GraphManager	163	90	no
BuildStrongCommand	166	72	no
Cluster	267	73	no
BuildGraphCommand	301	159	no

Figure 6: *Stub Summary: size and minimality.*

package for automatic differentiation of algorithms and COS is the *Class Ordering System* described in Section 4. The ep matrix test case is an extended precision matrix application that uses *NTL*, a high performance portable C++ number theory library [22]. vkey is a GUI application that uses the *V GUI* library [27], a multi-platform C++ graph framework for GUI applications. The final three test cases, IV Edraw, IV Graphdraw and IV Drawserv, are GUI applications generated from the *IV Tools* drawing application [26], a suite of free XWindows drawing editors for Postscript, TeX and web graphics production.

In the next section we describe the stubs that we use in all experiments reported in this paper followed by our study of coverage when a stub is used in place of a concrete class. We then describe our study of coverage when a stub is used as a direct implementation of an abstract base class and we conclude this section with a summary.

The Stubs

Figure 6 lists summary information about the stubs that we build for our study. The first column in the figure lists the names of the twelve classes in the cluster under study; the names are sorted by the lines of code, LOC, in the original class. The second column lists the LOC in the original class and the third column lists the LOC in the corresponding stub class. The final column of data in the figure, **Min**, reports "*yes*" if we were able to build a minimal stub according to the criteria established in Section 2, and "*no*" if the constructed stub violates one or more of the criteria.

The figure shows that seven of the twelve stubs are minimal and that some of the

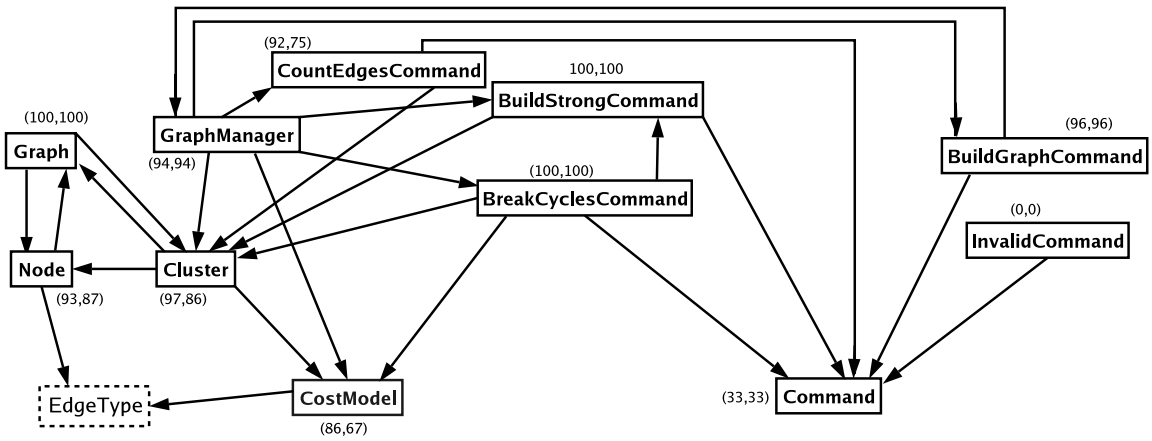


Figure 7: *An ORD for COS*. This figure shows statement coverage using a stub for class EdgeType.

stubs have the same number of LOC as the original class; for example, line 1 of the table shows that both the original and the stub for class InvalidCommand contained five lines of code. However, some stubs are much simpler than the original class; for example, line 5 of Figure 6 lists class BreakCyclesCommand, which was reduced by 67% from 61 LOC to 20 LOC.

We were unable to construct minimal stubs for five of the classes including BreakCyclesCommand, GraphManager, BuildStrongCommand, Cluster and BuildGraphCommand. Our criteria for stub construction requires that we simulate the original method using straight-line code and without invocation of other methods. We might relax this last criteria to permit invocation of methods within the stub but this relaxation is unlikely to enable further construction of meaningful stubs. Consider class BuildGraphCommand, which must simulate construction of an ORD. To build an ORD edges must be inserted into the graph; however, only the Graph hierarchy permits edge insertion. Thus, the stub for class BuildGraphCommand is unable to simulate the behavior of the original class. The criteria that stubs must consist of straight-line code further exacerbates stub construction difficulty. For example, class Cluster contains a transformer method RemoveLightestEdge, which must find and remove an edge or the program will loop forever. To find an edge, loop and decision structures are required to traverse the adjacency list of edges, violating the straight-line code criteria.

Code Coverage: Concrete Classes

Figure 7 illustrates an ORD for the COS application; recall that nodes in the ORD represent classes and the edges represent dependencies between the classes. The node, EdgeType, in the lower left corner of Figure 7 is a dotted rectangle indicating that, for this instance of an ORD, we are investigating coverage in the corresponding

```
( 1)  class Graph {
( 2)  public:
( 3)    Graph() {}
( 4)    Graph(const Graph & n) {}
( 5)    virtual ~Graph() {}
( 6)    virtual int size() const = 0;
( 7)    virtual void addEdge() = 0;
( 8)    virtual void deleteEdge() = 0;
( 9)  };
```

Figure 8: *Abstract Base Class.*

classes when using either the original class or a stub for the class, EdgeType, a leaf in the COS. Each of the rectangles in the ORD is annotated with an ordered pair (as described in Section 3). Recall that for an ordered pair, the first number is the percentage of statements covered using the original class Edgetype, and the second number is the percentage of statements covered using a stub for class Edgetype.

Considering the ordered pairs in the annotated ORD, we first observe that full code coverage is difficult to obtain when testing at the class cluster level. For example, consider classes Node and CostModel with pairs $(93, 87)$ and $(86, 67)$ respectively. Using the original class Edgetype only 93% and 86% of the statements were covered in the two clients of Edgetype, Node and CostModel respectively. Statements in the copy constructor and some exception handling statements account for the unexecuted statements in Node; similarly, some exception handling statements and some debugging statements explain the unexecuted statements in CostModel.

Moreover, statement level coverage degraded further in both clients of Edgetype when a stub was used in place of the original class. For example, the coverage in Node and CostModel degraded from 93% to 87% and from 86% to 67% respectively in the presence of a stub for Edgetype. More importantly, statement level coverage degraded for classes in the ORD that are not clients of Edgetype, namely classes Cluster and CountEdgesCommand, where the coverage degraded from 97% to 86% and from 92% to 75% respectively. This degradation of coverage in non-client classes is subtle. Finding this relationship between the stubbed class and the several times removed client is potentially quite difficult.

Code Coverage: Abstract Classes

The difficulty of testing abstract classes is well documented [3, 11, 15, 18, 25]. The main problem is that, for most languages that support object technology, abstract classes cannot be instantiated. Reference [15] describes an approach for computing an ordering of classes for testing in the presence of abstract classes; however, for the most part abstract classes are ignored in the ordering methodology literature.

Several strategies have been suggested for testing an abstract class. Most of these

Class name	EdgeType	Graph	Cluster	CostModel
InvalidCommand	0,0	0,0	0,0	0,0
Graph	100,100	-	100,100	100,100
Command	33,33	33,33	33,33	33,33
EdgeType	-	91,96	91,86	91,91
BreakCyclesCommand	100,100	100,100	100,100	100,100
CostModel	86,67	86,86	86,52	-
CountEdgesCommand	92,75	92,92	92,92	92,92
Node	93,87	93,93	93,80	93,93
GraphManager	94,94	94,94	94,94	94,94
BuildStrongCommand	100,100	100,100	100,100	100,100
Cluster	97,86	97,97	-	97,97
BuildGraphCommand	96,96	96,96	96,96	96,96

Figure 9: *Coverage Summary for four classes.*

strategies involve using a derived class, either an existing derived class or a stub, to enable testing of the abstract class [3, 18, 25]. Researchers have described the importance of re-testing a virtual base class method in the context of the derived class if that method invokes a re-defined method in the derived class [11, 25]. However, the focus of this paper is code coverage adequacy in the presence of stubs. Thus, to build a stub for an abstract class we adopt the approach described in reference [18] whereby we construct a direct implementation of a concrete version of the abstract class using a null body for any pure virtual member functions.

Figure 8 depicts an abstract class, Graph, written in C++, with a default constructor on line 3, a copy constructor on line 4, a virtual destructor on line 5, and three pure virtual functions on lines 6 through 8. Since Graph contains pure virtual functions, it is abstract and cannot be instantiated. To construct a stub for Graph using the *direct approach*, we supply empty bodies for the pure virtual functions size, addEdge and deleteEdge.

The COS application contains two abstract classes, Graph and Command. Using stubs for these two classes we found that code coverage did not degrade for any classes in the ORD even when we used stubs for both abstract classes in the same execution. Moreover, we found the two abstract classes to be the easiest to stub.

Summary: Coverage Adequacy for Cluster Testing with Stubs

Figure 9 summarizes coverage in cluster testing of four of the classes in the COS. The first column in the figure lists the twelve classes in the cluster and the second column lists ordered pairs of coverage for each of the twelve classes when a stub is used for class EdgeType. The third, fourth and fifth columns list ordered pairs of coverage for the twelve classes when a stub is used for classes Cluster, Graph and CostModel respectively. For example, the column with the vertical label EdgeType lists the ordered pairs attached to the nodes in Figure 7. The sixth row of data in

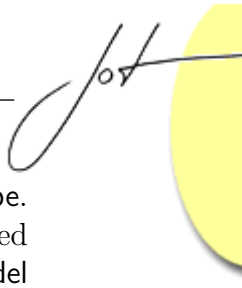


Figure 9 lists coverage in CostModel as $(86, 67)$ when a stub is used for EdgeType. Recall that $(86, 67)$ indicates that 86% of the statements in CostModel are covered when the original class EdgeType is used and 67% of the statements in CostModel are covered when a stub is used for EdgeType.

The second column with vertical label Graph summarizes coverage for the abstract class Graph. The third column with vertical label Cluster summarizes coverage for class Cluster, where we see degradation in statement coverage similar to that of EdgeType in column one of Figure 9. In contrast, for Cluster, none of the client classes suffered coverage degradation in the presence of stubs, including classes Graph, GraphManager, CountEdges, BuildStrongCommand, or BreakCyclesCommand. However, coverage degraded for other classes in the ORD including Node, EdgeType, and CostModel. The steepest degradation is in class CostModel, listed in row six of the figure, where coverage degraded from 86% to 52%. This degradation occurred in CostModel because the minimal stub for Cluster always identifies a node type as *association*, so that lines of code in CostModel for other kinds of edge types remain uncovered.

Recall from Section 2 that a primary requirement that we place on minimal stubs is that they enable the program to run to normal completion in cluster testing. However, to enable normal termination of the program in the presence of a stub for Cluster, we required loops and decision structures in three of the ten member functions. This inability to build a minimal stub for Cluster together with the difficulty that we faced in building stubs for other classes, such as BuildGraphCommand, BuildStrongCommand, and BreakCyclesCommand, correlates with other research regarding stub construction difficulty [15, 25].

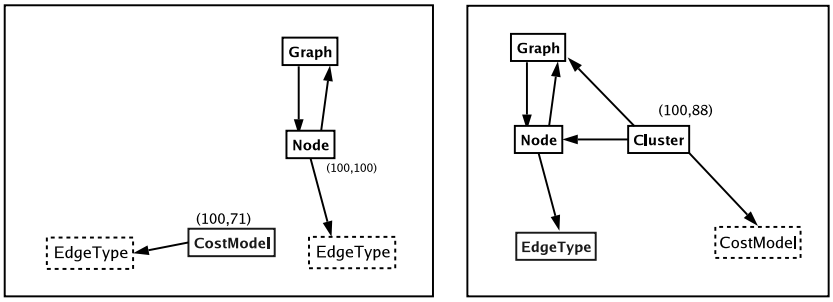
Based on this study, we conclude with four observations about cluster testing:

- Producing minimal stubs is the ideal, when that is possible (see the next item), but producing such stubs may be impossible. For these cases, the requirement that stubs may not call methods is too severe, and consideration should be given to allowing stubs to call other methods under specific conditions.

- When minimal stubs are possible they provide good statement coverage, often matching or nearly matching the "non-stubbed" coverage and never dropping below 60% of the coverage provided by the original supplier class.

- Coverage can degrade in a client class that uses a stub class, so that statements that were formerly covered using the original class remain uncovered in the presence of a stub.

- Subtle interactions may occur between classes in a cluster so that coverage can degrade in a class that is not directly dependent on a stubbed class or the client of a stubbed class.



(i) (ii)

Figure 10: *The graph on the left of the figure illustrates testing classes* CostModel *and* Node *using a stub for* EdgeType. *The graph on the right of the figure illustrates testing class* Cluster *using a stub for* CostModel.

6 RESULTS FOR CLASS-BASED TESTING

In this section, we study class-based testing and the effect of stubbing on coverage adequacy. Recall from Section 3, that we use the same ORD and stubs as used in cluster testing. In addition, for each class under test we: (1) write contracts for the methods of that class; (2) construct an MSG to generate message sequences for testing the class; and, (3) develop a test suite and write a driver. Finally, we stub each supplier in turn and use gcov to measure statement coverage in that class under test using the test suite specific to that class.

As an example of our approach and the results that we obtained we consider the examples in Figures 10(i) and 10(ii), which show the effect on the test coverage for Node and CostModel when EdgeType is stubbed, and on Cluster when CostModel is stubbed. Specifically, Figure 10(i) summarizes statement coverage in class CostModel, shown on the left side of the figure, and class Node, shown on the right side of Figure 10(i). The ordered pair attached to each node indicates the coverage when the original class EdgeType is used and when a stub for EdgeType is used. Note that in the context of class-based testing, to develop effective test cases, the tester must typically have intimate knowledge of the code not only for the class under test but for its supplier classes as well. Moreover, for most classes, multiple iterations through the generated message sequences are required to obtain full coverage.

Figure 10(i) shows that (using the test suites we developed) we were able to obtain full coverage in both CostModel and Node when using the original class EdgeType. However, as in cluster testing, coverage can degrade in the presence of a stub for a supplier class. For example, the coverage in CostModel degraded from 100% to 71%, as illustrated by the ordered pair attached to CostModel in Figure 10(i). In contrast, for class Node, coverage remained full when either the original supplier class, EdgeType, or a stub for EdgeType class is used to test Node.

Class Under Test	Reduce Ratio	Avg Reduce
InvalidCommand	0/1	0%
Graph	-	-
BreakCyclesCommand	1/2	0%
CostModel	1/1	29%
CountEdgesCommand	0/1	0%
Node	0/2	0%
GraphManager	2/2	27%
BuildStrongCommand	0/1	0%
Cluster	2/3	21%
BuildGraphCommand	0/1	0%

Figure 11: *Coverage Summary in Class-based Testing.*

Figure 10(ii) shows coverage in class Cluster relative to supplier class CostModel. The ordered pair, $(100, 88)$, shows that full coverage was obtained when all of the original supplier classes were used but coverage degraded to 88% when a stub was used for CostModel.

While Figure 10(ii) shows the result for Cluster using a stub for CostModel, recall that in our experiments, we actually stubbed each supplier class in turn. Relative to the coverage in Cluster this means that we also stubbed (one at a time) Node and Graph. In those results (not illustrated here due to space constraints), the coverage for Cluster degraded to 86% when using a stub for Node but remained at 100% when using a stub for Graph. This result highlights the importance of edge choice in removing cycles in an ORD for class-based testing.

Figure 11 presents a summary of the coverage in each of the classes that we test in the COS. The first column of the table in that figure, Class Under Test, lists the classes in the COS. Note that EdgeType and Command are not listed since they are leaf classes in the ORD, having no supplier class (hence, they are totally unaffected by the stubbing of other classes). The second column, Reduce Ratio, lists the ratio of the number of supplier classes for which coverage was reduced when a stub is used, in turn, compared to the total number of suppliers that each class uses for which we were able to construct a minimal stub (see Section 5). The final column, Avg Reduce, lists the average reduction in coverage when a stub is used for each of the minimal stubs. For example, the next-to-last row of Figure 11 lists class Cluster, where the table shows that coverage was reduced in 2 of the 3 supplier classes for which we were able to construct a minimal stub; the three suppliers are Node, CostModel and Graph and coverage was reduced in the presence of stubs for Node and CostModel. The final column for this row shows that the average reduction in coverage when a stub is used for Node and CostModel was 21%. The table lists a *dash* for the coverage reduction of Graph, which is abstract so we did not test it.

From this study we have the following observations about class-based testing:

- Coverage can degrade in a client class that uses a stub class, so that statements that were formerly covered using the original class remain uncovered in the presence of a stub.
- When minimal stubs are possible they provide good statement coverage. In our experiments, coverage is reduced in only 6 of 14 class test executions, and the reduction in coverage is typically modest.
- When breaking cycles in an ORD for class-based testing, the choice of which edge to remove (hence which supplier class to stub), has an impact on the *adequacy* of coverage in client classes in class based testing.

7 CONCLUSIONS

We presented an approach to investigating code coverage in the presence of stubs when they are used in cluster or class-based testing. Along with giving a specific definition of a *minimal stub*, we provided a classification of the kinds of methods that are candidates for stubs in an object-oriented application. We presented a case study where we examined the adequacy of test coverage in the presence of stubs for a particular application (COS). Two important conclusions of the study are: First, for both cluster and class-based testing in the presence of stubs, coverage can degrade in a client that uses a stub class in place of the original supplier class, so that statements that were covered when the original supplier class is used remain uncovered in the presence of a stub for the supplier. This result highlights the importance of edge choice for removal considerations in the presence of cycles in an ORD. Second, subtle interactions may occur between classes in a cluster so that coverage can degrade in a class that is not directly dependent on a stubbed class or on the client of a stubbed class. These conclusions have important consequences on software security and testing [28].

REFERENCES

[1] I. Bashir and A. Goel. Testing C++ classes. *Software Testing, Reliability and Quality Assurance*, pages 43–48, 1994.

[2] S. Best. Analyzing code coverage with gcov. *Linux Magazine*, pages 43–50, July 2003.

[3] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley, 2000.

[4] Boldsoft, Rational Software Corp, IONA and Adaptive Ltd. Response to the UML 2.0 OCL RfP. Technical report, OMG Document ad/2002, March 1 2002.

[5] L. Briand, Y. Labiche, and Y. Wang. Revisiting strategies for ordering class integration testing in the presence of dependency cycles. *ISSRE*, pages 287–297, 2001.

[6] U. Buy, A. Orso, and M. Pezze. Automated testing of classes. *ISSTA*, pages 39–48, 2000.

[7] H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen. In black and white: An integrated approach to class-level testing of object-oriented programs. *ACM TOSEM*, 7(3):250–295, July 1998.

[8] P. Clarke and B. A. Malloy. A unified approach to implementation-based testing of classes. *ICIS*, pages 226–234, 2001.

[9] R. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM TSE*, 3(2):101–130, 1994.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[11] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick. Incremental testing of object-oriented class structures. *ICSE*, pages 68–80, 1992.

[12] M. J. Harrold and G. Rothermel. Peforming data flow testing on classes. *SIGSOFT FSE*, pages 154–163, 1994.

[13] S. Kirani and W. T. Tsai. Method sequence specification and verification of classes. *Testing Object-Oriented Software*, pages 43–53, 1998.

[14] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. A test strategy for object-oriented programs. *COMPSAC*, pages 239–244, 1995.

[15] Y. Labiche, P. Thevenod-Fosse, H. Waeselynck, and M. H. Durand. Testing levels for object-oriented software. *ICSE*, pages 136–145, 2000.

[16] Y. Le Traon, T. Jeron, J.-M. Jezequel, and P. Morel. Efficient object-oriented integration and regression testing. *IEEE Trans. on Reliability*, 49(1):12–25, 2000.

[17] B. A. Malloy, P. J. Clarke, and E. L. Lloyd. A parameterized cost model to order classes for class-based testing of C++ applications. *ISSRE*, pages 353–364, 2003.

[18] J. D. McGregor and D. A. Sykes. *A Practical Guide To Testing Object-Oriented Software*. Addison-Wesley, 2001.

[19] G. C. Murphy, P. Townsend, and P. S. Wong. Experiences with cluster and class. *CACM*, 37(9):39–47, 1994.

[20] NIST. The economic impacts of inadequate infrastructure for software testing. Technical Report, May 2002.

[21] D. E. Perry and G. E. Kaiser. Adequate testing and object-oriented programming. *Testing Object-Oriented Software*, pages 5–10, 1998.

[22] V. Shoup. Number theory library. http://www.shoup.net/ntl/, March 2002.

[23] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[24] K.-C. Tai and F. J. Daniels. Test order for inter-class integration testing of object-oriented software. *COMPSAC*, pages 602–607, 1997.

[25] N. N. Thuy. Testability and unit tests in large object oriented software systems. *Fifth International Software Quality Week*, 1992.

[26] J. M. Vlissides and M. A. Linton. IV tools. http://www.vectaport.com/ivtools/, March 2002.

[27] B. Wampler. The V C++ GUI framework. http://www.objectcentral.com, October 2001.

[28] J. A. Whittaker and H. H. Thompson. *How to Break Software Security: Effective Techniques for Security Testing*. Addison-Wesley, 2004.

ABOUT THE AUTHORS

Errol Lloyd is a Professor of Computer and Information Sciences at the University of Delaware. Previously he served as a faculty member at the University of Pittsburgh and as Program Director for Computer and Computation Theory at the National Science Foundation. From 1994 to 1999 he was Chair of the Department of Computer and Information Sciences at the University of Delaware. Concurrently, from 1997 to 1999 he was Interim Director of the University of Delaware Center for Applied Science and Engineering in Rehabilitation. Professor Lloyd received undergraduate degrees in both Computer Science and Mathematics from Penn State University, and a PhD in Computer Science from the Massachusetts Institute of Technology. His research expertise is in the design and analysis of algorithms. In 1989 Professor Lloyd received an NSF Outstanding Performance Award, and in 1994 he received the University of Delaware Faculty Excellence in Teaching Award.

Brian A. Malloy is an associate professor in the department of Computer Science at Clemson University. Brian's research focus is software engineering and compiler technology. He has investigated issues in software validation, testing and program

representations to facilitate validation and testing. He has applied software engineering to parser development, especially as applied to the construction of a parser and front-end for C++. In addition, he has developed techniques to validate contracts for C++ applications, including the development of an extended form of class invariants, temporal invariants. He has investigated issues in class-based testing and has developed a parameterized cost model to provide an integration order for class-based testing of C++ applications. Brian has also been active in software visualization, especially applied to visualization of UML models. malloy@cs.clemson.edu.
See also http://www.cs.clemson.edu/∼malloy.