A Parameterized Cost Model to Order Classes for Class-based Testing of C++ Applications

Brian A. Malloy
Computer Science Department
Clemson University
Clemson, SC 29634, USA
malloy@cs.clemson.edu

Peter J. Clarke
School of Computer Science
Florida International University
Miami, FL 33199, USA
clarkep@cs.fiu.edu

Errol L. Lloyd
Computer & Information Sciences
University of Delaware
Newark, DE 19716, USA
elloyd@udel.edu

Abstract

In this paper we present the design and implementation of a Class Ordering System that is driven by a parameterized cost model. The parameters to the model assign weights to the edge types that describe the relationships between the classes in the graphical representation of the program. The nodes in the graph are classes and the edges express relationships between the classes. Previous research has included three or four edge types in the graph. However, to accommodate the full complement of C++ language constructs, which include template classes and functions and nested classes, we extend the graph to include six edge types. The parameters to the cost model can be tuned to remove certain types of edges in an attempt to reduce the cost of the testing effort or to reduce the cost of breaking cycles in the graph. Our case study indicates that inclusion of inheritance edges in cycle breaking considerations may reduce the number of edge removals by a factor of two or more.

1. Introduction

As software developers shift their priorities to the construction of complex, large scale systems that are easy to extend, modify, and maintain, the object-oriented approach becomes more attractive than traditional methodologies. The cost of system nonperformance and failure is expensive, with sometimes catastrophic impact [20]. Therefore, the trend in the development of these large systems has shifted toward testable, robust models, whose focus is on preventing errors. One process that supports the construction of robust software is testing. An advantage of software testing is the relative ease with which some of the testing activities can be performed, such as executing the program using a given set of inputs, or test cases, and then comparing the generated output to the expected output.

However, testing a complete object-oriented system presents some imposing problems. For example, the entire system must be available before the test can begin and this may be very late in the life cycle. During the test of a complete system there are risks of complex interactions among the errors and of mutual destabilization of the corrected classes or components. Thus, many developers prefer a progressive approach where the first stage consists of testing individual classes. However, classes interact with other classes so that a fundamental issue in testing object-oriented systems is to determine an integration order for the classes. The goal is to find the best order to test classes to avoid or reduce the construction of stubs for untested classes.

To determine an order for class-based testing, previous approaches have constructed an object relation diagram, ORD, whose nodes are classes and whose edges are relationships between the classes [14]. If there are no cycles in the ORD, then a reverse topological ordering of the nodes will yield a test order that obviates the construction of stubs.

However, in the presence of cycles in the ORD, one or more edges must be removed and, 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. The problem of removing a minimum number of edges to eliminate cycles in an ORD is equivalent to the *feedback arc set*, which has been shown to be NP-complete [10, 13]. Thus, previous research has focused on the development of heuristics that attempt to balance the cost of breaking cycles with the cost of constructing stubs to enable testing of client classes that use untested supplier classes [7, 14, 25].

In this paper we present the design and implementa-

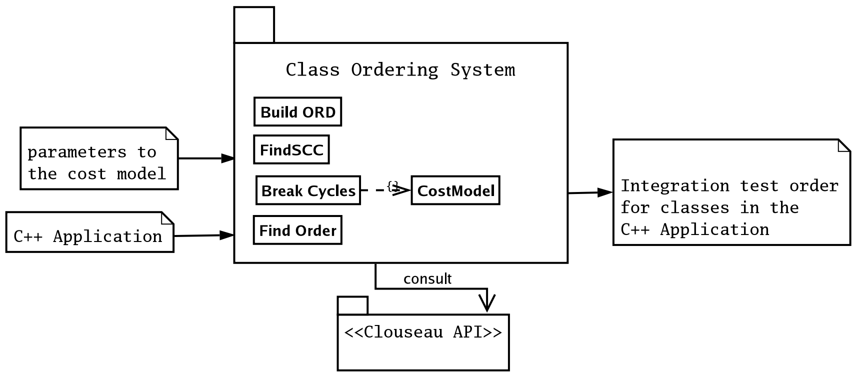


Figure 1. *Overview of our Class Ordering System.*

tion of a *Class Ordering System* that is driven by a parameterized cost model. The parameters to the model assign weights to the edge types that describe the relationships between the classes in the ORD. Previous research has included three [14, 25] or four [6, 15] edge types in the ORD. However, to accommodate the full complement of C++ language constructs, which include template classes and functions and nested classes, we have extended the ORD to include six edge types. The parameters to the cost model can be tuned to remove certain types of edges in an attempt to reduce the cost of the testing effort (i.e., the number of stubs that are required to test the program) or to reduce the cost of breaking cycles in the ORD.

Our results indicate that ORDs for C++ applications contain few cycles but that the cycles can include hundreds of classes and thousands of edges. Our results suggest further that increasing the efficiency of the cycle breaking computation may also increase the number of stubs that a tester might be required to construct. A better strategy entails the use of parameters to the cost model that might increase the cost breaking computation, but will decrease the number of required stubs and thereby increase the efficiency of the testing effort.

In the next section, we provide an overview of our approach and in Section 3 we motivate and describe the complexity of stub construction. In Section 4 we describe our ORD and in Section 5 we present our *Class Ordering System* for computation of an order for class-based testing. In Section 6 we describe the results of our case study and in Section 7 we review research that relates to our work. Finally, in Section 8 we draw conclusions.

2. Overview

In this paper we present a *Class Ordering System* that is driven by a cost model that estimates the cost of the testing effort. Our goal is to tune the parameters to the model to measure the tradeoff between achieving efficiency in the cycle breaking computation as compared to achieving efficiency in the testing effort. In this section we provide an overview of the system.

Figure 1 provides an overview of our *Class Ordering System*. Inputs to the system are the parameters to the cost model and a C++ application for class-based testing. We compute an ordering of classes for testing in four steps. The first step entails the construction of an ORD for the application using six edge type designations that describe the relationships between the classes in the application.

If there are cycles in the ORD, in the second step we partition the ORD into strongly connected components. In the third step, we use the parameterized cost model to compute the weights of each edge in the strong components, based on the type designation of the edge; edges with the same source and destination class are then merged. In the third step, edges with the smallest weights are removed from the strong components until there are no cycles. Finally in the fourth step, a reverse topological sort of the nodes in the ORD yields an integration test order.

To build an ORD for the C++ application under test, the source code for the application must be parsed, and variable and type information extracted from the application. To parse our application we use *keystone*, an ISO conformant parser and front-end [17] for the C++ language [12, 24]. *Keystone* includes the *Clouseau* Application Programmer's Interface, API, which provides the neces-

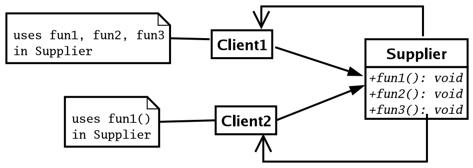


Figure 2. *Motivation for stub construction.* **The UML class diagram in this figure illustrates a cycle of dependencies between a supplier class and two client classes. We use this example to motivate the construction of stubs.**

sary information required to build an ORD for the application.

3. Stubs

The construction of stubs for classes is a complicated aspect of class-based testing [7, 14, 15, 25, 26]. Stubs are used in testing to facilitate decoupling of system modules and to enable development of cooperating modules before all modules are complete [18]. For our work, we define two kinds of stubs: *member function stub* and *class stub*. A *member function stub* is a dummy function that simulates some or all of the functionality of a real function in a class. A *class stub* is a dummy class consisting of member function stubs for some or all of the member functions in a real class.

In the next section we motivate the construction of stubs with a simple example. In Section 3.2 we discuss the complexity of stub construction in the presence of inheritance.

3.1. Motivation for stubs

Stubs must be constructed for an untested supplier class to enable testing of a client class that uses the supplier class. Consider the classes illustrated in Figure 2 with cycles of dependencies between classes Client1 and Supplier and Client2 and Supplier. Assume further that we break the cycles by removing the edges from Client1 to Supplier and from Client2 to Supplier; thus, Client1 and Client2 become leaf classes in a reverse topological sort of the graph and they should be tested before Supplier. However, to test class Client1 we must construct member function stubs for fun1, fun2, and fun3 and a class stub for Supplier, since they are used by Client1.

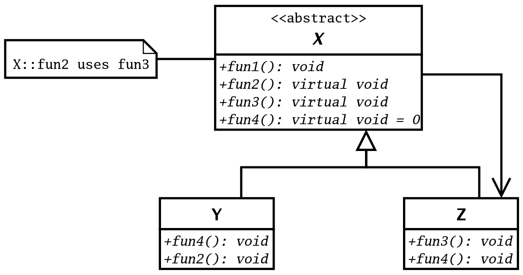


Figure 3. *Stub for an abstract base class.* **The UML class diagram in this figure illustrates an abstract base class and two derived classes that override some of the functions in the base class. We use this example to describe the difficulty of stub construction for abstract base classes.**

The efficiency of our edge removal strategy is increased if we can reuse the stubs for Supplier to test Client2. However, stubs must be simple so that they themselves are not error prone [3]. In fact, stubs should only contain sequential control flow to reduce the testing effort of stubs [7]. Therefore, the member function stub for fun1 may be inadequate for reuse in testing Client2 so that a new member function stub may be required for fun1 and a new class stub for Supplier. Thus, any dependency edge that is removed is likely to require stub construction for the corresponding supplier class and the number of stubs for a supplier class is usually proportional to the number of clients of that supplier [7].

3.2. Stubs in the presence of inheritance

Figure 3 illustrates an inheritance hierarchy with an abstract base class, X, and derived classes Y and Z. There is a cycle among classes X and Z. If we break the cycle by removing the inheritance edge from Z to X, then stubs must be constructed for X so that Z can be tested. Execution based testing of classes requires construction of an instance of the class; however, programming language semantics usually preclude the instantiation of abstract classes such as X. Several approaches to solving this problem have been suggested in the literature including the construction of a concrete subclass of X solely for the purpose of testing X [4, 26].

The problem of constructing stubs for an abstract base class can be avoided by removing an edge that is not involved in an inheritance relationship. Kung et al. recommend removing an association edge from an ORD to break cycles since, in C++, there is guaranteed to be an association edge in every cycle [14]. Unfortunately, this is not valid for programs that include template classes. In fact, the *curiously recurring template pattern* consists of a cycle of dependencies where the cycle does not include an association edge [27]. However, Kung et al. did not include template classes in their ORD construction.

Briand et al. use a genetic algorithm and coupling metric in an attempt to break cycles by removing edges that will reduce the complexity of stub construction [6]. They conclude that composition and inheritance relationships should never be removed since, according to their heuristic, removal of these edges would likely lead to complex stubs. The complexity of stub construction for parent classes is induced by the likely construction of stubs for most of the inherited member functions [7]; moreover, inherited member functions must be tested in the new context of the derived class rather than the context of the parent class [11, 26].

Figure 3 illustrates the complexity of testing base class member functions. Class X has a virtual function, fun2, which uses virtual function fun3. However, when fun2 is applied to an instance of Z it is applied to Z::fun3, rather than X::fun3. Therefore, a test of fun2 on an instance of X does not guarantee that fun2 is correct for an instance of Z [11, 26].

As a final illustration of the difficulty of testing a base class, consider class Y in Figure 3, which inherits fun1 and fun3. A test of fun1 and fun3 in Y might serve to also test fun1 and fun3 in X. However, to simplify test management it is preferable to test all member functions in a single class that redefines none of them, if such a class exists; otherwise a class stub should be constructed for this purpose [26]. Since class Y does not test fun3, to satisfy the "test all member functions in the same class" dictum, a class stub for X is required.

4. Graph Representation

In this section we describe the program representation that we use to find a class ordering for class-based testing. We use a variant of an Object Relation Diagram, ORD, used in previous research [14, 15, 25]. Our ORD is extracted by reverse engineering the source code of a C++ application using the Clouseau API [19]. An ORD[1] is a directed graph whose nodes are classes and whose edges represent the relationships between the classes.

[1]The use of the term ORD is a bit of a misnomer, since the nodes are classes, not objects; however, since the term is used in previous research, we continue its use in this paper.

4.1. Edge type designations

The ORD described in references [14, 25] uses three types of edges, and reference [15] extends the ORD to include a fourth type of edge. However, the focus of our work is the analysis of existing C++ applications, including template functions and classes and nested classes; thus we require six edges in our ORD including the addition of *ownedElement* and *composition* edges. We use the UML specification for these edges [5].

The edges in our ORD capture relationships between the classes in the program under test and are specified by the syntax and semantics of the data attributes of classes and the parameters or local variables of member functions. The six types of edges in our ORD are *association*, *composition*, *dependency*, *inheritance*, *ownedElement* and *polymorphic* edge. The first five types of edges are used in UML class diagrams and we base our use of these edges on the UML specification, version 1.5 [21]. The *polymorphic* edge is presented in reference [15] as a dynamic edge.

The meaning of *inheritance*, *ownedElement* and *polymorphic* edges are fairly straightforward and are described in the example of Section 4.2. However, there is some disagreement about the meaning of the other edges. In our ORD for C++ applications we use a *composition* edge for a class data attribute whose lifetime is bound to the lifetime of the containing object. We use an *association* edge for a class data attribute that is a reference or pointer to another class. Reference [22, page 2-33] states that a *dependency* is "a term of convenience for a relationship other than association and generalization" and "the client requires the presence of the supplier". We use a *dependency* edge for a parameter or local variable of a member function. Therefore, both association edges and dependency edges may generate polymorphic edges.

The most controversial distinction among edges is between aggregation and association [8, page 85]. References [21, page 2-9] and [22] both state that *aggregation* is *association* and that "the distinction between aggregation and association is often a matter of taste" [22, page 148]. Thus, we choose to use the more general relationship, association, and we do not use an aggregation edge in our ORD.

4.2. A sample ORD

Figure 4 contains a C++ program and Figure 5 illustrates a corresponding ORD for the program. A template class, W, is illustrated on lines 1 through 7 of Figure 4. There is no direct representation of a template class in our ORD since a template class represents a partial specification of a class whose arguments are usually other

```
( 1)  template <class T>
( 2)  class W {
( 3)  public:
( 4)     void fun(T t) {}
( 5)  private:
( 6)     T data;
( 7)  };
( 8)  template <>
( 9)  class W<char> {
(10)     char* a;
(11)  };
(12)  class P;
(13)  class X {
(14)     W<P*> w;
(15)  };
(16)  class P {
(17)  public:
(18)     void fun(X x) {}
(19)  private:
(20)     W<char> wp;
(21)  };
(22)  class R;
(23)  class Q : public P {
(24)  private:
(25)     R* r;
(26)  };
(27)  class R : public P {};
(28)  class S : public R {};
(29)  class Y : public X {};
(30)  class Z : public X {
(31)        class Inner {};
(32)  };
```

Figure 4. *Sample C++ program.*

classes. A template class is not directly executable and, thus, not directly testable. A template class becomes a class when it is instantiated with actual arguments supplied for the formal template parameters. An instance of template class W is listed on lines 8 through 11 of Figure 4 and illustrated in the upper right corner of Figure 5. Class W<**char**> is a *template specialization* with the template formal argument specified as **char**.

Line 12 of Figure 4 lists a forward declaration of class P and lines 13 through 15 list class X containing data attribute w, an instance of template class W with actual parameter P*. We illustrate class W<P*> with a box in the middle of Figure 5. Class W<P*> has a data attribute, data, that is a pointer to P since the template class is instantiated with P*; we illustrate this relationship between W<P*> and P by the *association* edge connecting the two classes. The *association* relationship indicates that W<P*> has a relationship with a data object; however, this relationship can be severed before destruction of the W<P*> object or transferred through pointer assignment. The association relationship is therefore not as strong as the *composition* relationship,

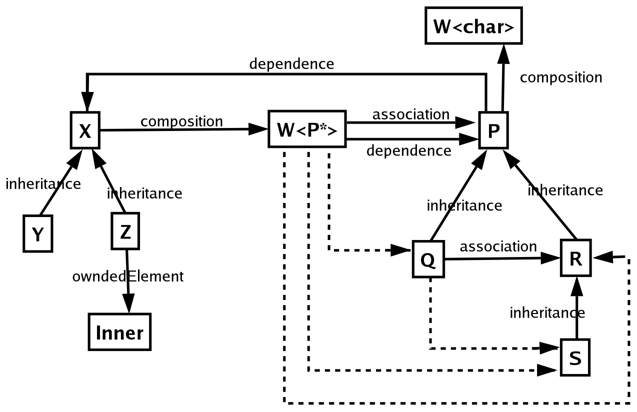


Figure 5. *ORD for sample C++ program*. **This figure illustrates an ORD for the program listed in Figure 4, and the edges that capture relationships between classes in the program. The four dashed lines in the graph are** *polymorphic* **edges, and the other edges are labeled appropriately.**

which cannot be severed until the object that owns the data is destroyed. Data attribute w in class X, line 14 of Figure 4, illustrates a *composition* relationship represented by the arrow from X to W<P*> on the middle left of Figure 5. The ownership of w by class X cannot be severed until class X is destroyed.

Since class W<P*> contains data, a pointer to P, variable data can polymorphically refer to P or any of its derived classes and an invocation to a virtual function in P will be resolved dynamically. Thus, the *association* edge connecting W<P*> to P generates polymorphic edges from W<P*> to each of the classes derived from P, illustrated as dashed lines from W<P*> to Q, R and S. Polymorphic edges are illustrated as unlabeled dashed lines in our ORD and were described in reference [15], where they were referred to as dynamic edges; however, since these edges can be determined statically, we refer to them as *polymorphic* edges.

Lines 16 through 21 of Figure 4 list class P containing member function fun and data attribute wp. Formal parameters and local variables of functions form *dependency* relationships between classes; thus, there is a dependency edge from class P to class X induced by parameter x of function fun, line 18 in Figure 4. This parameter is passed by value and therefore does not generate polymorphic edges from class P to the classes derived from class X; if the transmission mode were reference or if parameter x was a pointer to X, then polymorphic edges would be generated from class P to classes Y and Z. Also class P has lifetime ownership of data attribute wp, expressed as the composition edge to the template specialization W<**char**> in the upper right of Figure 5.

Lines 23 through 26, of Figure 4, list class Q, derived publicly from class P. This *inheritance* relationship is illustrated in Figure 5 by the edge from Q to P labeled *inheritance*. Since class Q has an association with class R, generated by the pointer variable on line 25 of Figure 4, there is an association edge from Q to R and a polymorphic edge generated from class Q to class S, derived from class R. There are similar inheritance edges in Figure 5 for the derived classes listed on lines 27 through 32 of Figure 4.

Finally, class Inner, listed on line 31 of Figure 4 is contained in class Z, generating an ownedElement edge from Z to Inner, illustrated on the lower left of Figure 5.

4.3. The parameterized cost model

The usual approach to the computation of an integration order for classes in the presence of a cycle of dependencies among the classes, is to construct an ORD and remove edges from the ORD until all cycles are broken. One approach might entail the removal of arbitrary edges from the ORD until all cycles are eliminated. However, our approach is to use our *Cost Ordering System*, a flexible framework that is driven by a cost model that assigns weights to the edges of an ORD. The parameters to the model provide flexibility in guiding the edge removal process so that the model can be tuned in an attempt to minimize the number of edges that are removed or to minimize the number of stubs that are required for untested supplier classes.

Our ORD is a multigraph G = (V, E), where V is a set of vertices representing classes, and E is a set of edges representing the relationships between the classes. A multigraph may contain multiple edges between any particular pair of vertices

Our cost model, $C = <\mathcal{W}, f(e), w(m_{x,y})>$, is a 3-tuple consisting of $\mathcal{W}$, a set of weight assignments and functions $f(e)$ and $w(m_{x,y})$ defined as follows:

$$\mathcal{W} = \{w_1, w_2, w_3, w_4, w_5, w_6\} \qquad (1)$$

$$f \; : \; E \rightarrow \mathcal{W} \qquad (2)$$

$$for\ a\ given\ x, y \in V, \;\; m_{x,y} = \{(x, y) \in E\} \qquad (3)$$

$$w(m_{x,y}) = \sum_{e \in m_{x,y}} f(e) \qquad (4)$$

Equation (1) is a set of weight assignments for the six edge type designations for inheritance, association, composition, dependence, polymorphic and ownedElement edges. Equation (2) defines a total function f as a mapping from the set of edges, E, to the set of weights $\mathcal{W}$, so that for edge e, $f(e)$ is the weight assignment for that edge. Equation (3) defines a merged edge $m_{x,y}$ as a set of edges represented as ordered pairs (x, y), where each edge in the set has the same source class and the same destination class. Equation (4) defines $w(m_{x,y})$, a function w that computes the weight of a merged edge $m_{x,y}$ as the sum of the weights of the individual edges in the set $m_{x,y}$.

5. The Class Ordering System

In this section, we describe our algorithm to order the classes for class-based testing. In Section 5.3 we describe the design of our system.

5.1. The algorithm

Figure 6 summarizes the algorithm used by our *Class Ordering System* to order the classes in a C++ application for class-based testing. In *Step 1* we build an ORD using variable and type information garnered from the Clouseau API in the *keystone* parser and front-end [17]. In item *(1)* of *Step 1* we assign weights to each edge in the ORD using function f of our cost model, equation (1) in Section 4.3. In item *(2)* of *Step 1* we merge edges using $w(m)$, equation (4) in Section 4.3. Assume classes X and Y with directed edges (x_1, y_1) and (x_2, y_2) connecting X and Y respectively. We merge the two directed edges (x_1, y_1) and (x_2, y_2), label the merged edge as *merged*, and assign a weight to the new edge that is the sum of the weights of (x_1, y_1) and (x_2, y_2). To complete the merge process of item *(3)* we merge all such edges.

In *Step 2* we partition the nodes of the ORD into strongly connected components, SCC, using depth first search as described in reference [2].

In *Step 3* we use our cost model to break the cycles in SCCs constructed in *Step 2*. We begin with item *(1)* of *Step 3* in Figure 6 where we first choose a SCC, c, that has more than one node. Strongly connected components with a single node X either have association or dependency edges starting and ending with X or no edges at all. Such classes will require no stub construction during the testing cycle. Finding the best edge to remove is equivalent to finding a solution to the *feedback arc set* problem [10, p. 192] so we use the cost model to guide our decisions about edge choices. In item *(2)* of *Step 3* we remove the edge and in item *(3)* we use *Step 2* to find

```
Step 1: Build ORD, G, using info from Clouseau
    (1) Use function $f$, equation (2) in Section 4.3,
        to assign a weight to each edge in the ORD
    (2) merge the edges in G; use $w(m_{x,y})$, equation (4)
Step 2: Partition G into SCCs
Step 3: Use the cost model to break all of the cycles
        in each SCC as follows:
    (1) Choose a SCC, c, with more than 1 node
    (2) Remove an edge in c with smallest weight
    (3) Use Step 2 to find SCCs in the reduced SCC
    (4) Repeat Step 3 until all SCCs contain 1 node
Step 4: Find a class ordering for testing as follows:
    (1) For each edge e removed in Step 3, remove e
        from G; let resulting graph be G'
    (2) Order the classes in G' in reverse topological
        order
```

Figure 6. *Algorithm summary*. **This figure summarizes the steps required to build an ORD for a program, break the cycles in the ORD and find a class order for class-based testing.**

the strongly connected components in the reduced SCC. We repeat *Step 3* until all SCCs contain a single node.

In item *(4)* of *Step 3* we remove the edge with smallest weight, and use *Step 2* to find the cycles in c. We continue with steps 3 and then 2 until there are no more cycles, saving the removed edges in a list for processing in *Step 4*. In *Step 4* we remove the edges from G that were removed in *Step 3* and then find a class order using a reverse topological ordering of the nodes in the reduced graph G'.

5.2. Running time of the algorithm

The running time of the algorithm used by our *Class Ordering System* is $O(e*(n+e))$, where n and e are, respectively, the numbers of nodes and edges in the ORD. Specifically, the construction of the ORD (*Step 1*), finding the initial strongly connected components (*Step 2*), and producing the class ordering (*Step 4*), each require $O(n+e)$ time. *Step 3* has at most e iterations, with finding the resultant strongly connected components being the dominating step. Since finding those SCCs requires time $O(n+e)$ in the worst case, the total time for *Step 3* (and for the algorithm as a whole) is $O(e*(n+e))$.

5.3. Design of the system

Figure 7 is a UML class diagram that summarizes the important classes and relationships in our *Class Ordering System*; other classes and relationships in the system have been elided from the diagram for readability. The three classes at the top of the figure, Graph, Node and Cluster, are a variation of the Composite design pattern and are used to form the nodes and clusters in our ORD. The class at the middle left of the figure, GraphManager, is a version of the Singleton design pattern and choreographs the steps of the algorithm in Figure 6. The inheritance hierarchy at the bottom of the figure consisting of the five classes, Command, BuildGraphCommand, CountLinksCommand, BuildStrongComponentsCommand, and BreakCyclesCommand, are a version of the Command design pattern and encapsulate the functionality of the steps of the algorithm. The Clouseau API is shown in the lower left of the figure and the class that summarizes our 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 7; EdgeTypes encapsulates the types of edges in our ORD, described in Section 4.

The three classes at the top of Figure 7, Graph, Node and Cluster, form a variation of the Composite design pattern [9]. The Graph base class has two purely virtual functions, insert and addEdge, which all derived classes must implement. The Node derived class, shown in the upper left of Figure 7, has four data attributes: name, the name of the class that corresponds to this node, theType, the type of edge that is drawn to this class, visited, used in depth first search, and dfsNumber, also used in depth first search in *Step 2* of the algorithm in Figure 6.

Class Cluster, also derived from Graph, has a single data attribute, adjList, an adjacency list of nodes. We fully exploit the standard C++ library for *list*, *sort*, *find* and other functionality in our *Class Ordering System*, so our adjacency list is a $list < list < Node >>$. Class Cluster includes member functions, insert, to add a node to the list, addEdge, to add an edge to the list, getAdgList, to get the list, removeLightestWeight, to remove the edge with the smallest weight, setAllWeights, which uses the cost model to set the weights of the edges in the list, and mergeEdges, which merges the edges as described in Section 4.

The GraphManager class in the middle left of Figure 7, is a singleton, providing global access to a single instance of the class. The Singleton obviates the need to pass the data in the system to instances of Command. The figure illustrates two of the data attributes of GraphManager, graph, the ORD built by an instance of BuildGraphCommand, and cycles, a list of clusters, some of

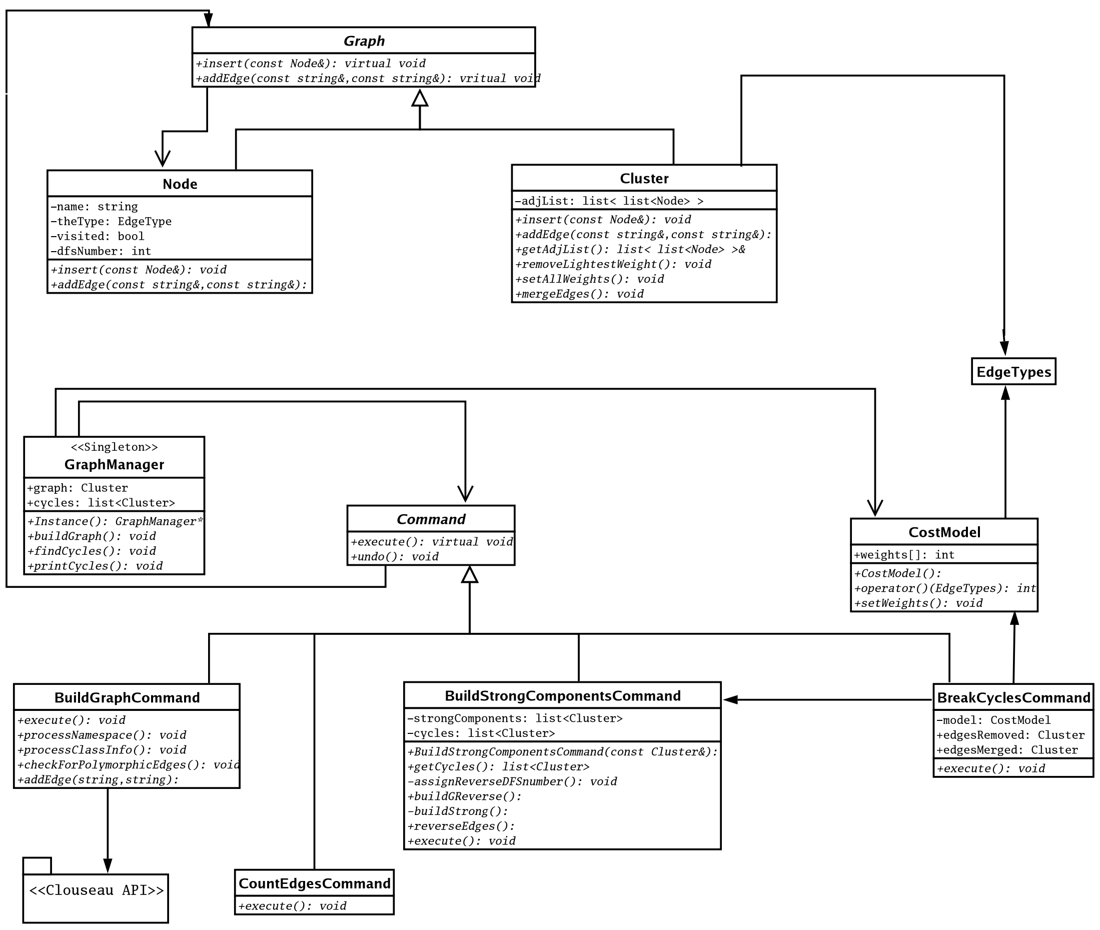


Figure 7. *Class Ordering System*. **This figure illustrates a UML class diagram summarizing the important classes and relationships in our system. The** CostModel **class in the middle right of the figure encapsulates our parameterized cost model. There is a cycle in the class diagram that includes classes** Graph**,** Node **and** Cluster**.**

which contain cycles larger than a single node. We have also listed four of the member functions of GraphManager including Instance, which provides global access to the GraphManager object, and buildGraph, findCycles and printCycles, which use instances of the command pattern.

The two classes in the lower right of the figure, BuildStrongComponentsCommand and BreakCyclesCommand, encapsulate the functionality of *Step 2* and *Step 3* respectively of Figure 6. The dependence edge from BreakCyclesCommand to BuildStrongComponentsCommand indicates the use of an instance of BuildStrongComponentsCommand described in item *(4)* of Step 3.

6. Case study

In this section we describe results of our implementation of the *Class Ordering System*, COS, executed on a $Dell Precision^{TM}$ 530 workstation with $Intel^{©}$ $Xeon^{TM}$ 1.7 GHz processor equipped with 512 MB of RDRAM, running the Red Hat Linux 8.0 operating system. The COS consists of 38 classes and 1,304 lines of C++ code [24], compiled with GNU *gcc* version 3.2.

In the next section we describe the test suite for our study. In Section 6.2 we provide results about the number of nodes and edges in the strongly connected components, SCCs, and in Section 6.3 we provide results about number of classes and edges in the largest SCC. Finally, in Section 6.4 we provides results about the number of

Test case	LOC	classes	template or nested classes
Adol-C	699	16	0
Class Ordering System (COS)	1304	38	11
ep matrix	4,944	50	0
vkey	8,588	46	0
IV Edraw	832	44	0
IV Graphdraw	4,354	151	1
IV Drawserv	5,687	236	1

Figure 8. *Test suite for our study*. **A test case in our study is a program that is used as input to our Class Ordering System, COS. We use the COS to build an ORD for the test case and then investigate issues about the ORD such as the number of strongly connected components (SCCs) in the ORD and number of edges that must be removed to eliminate cycles.**

edges that must be removed to break cycles in the SCCs using different weights for inheritance edges.

6.1. The test suite

The table in Figure 8 summarizes our suite of seven test cases, listed in the rows of the table as 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 COS to investigate issues such as the number of cycles in the ORD and the number of edges removed to eliminate cycles in the ORD for the test case. The test cases were chosen for their range and variety of application. The test cases are mostly listed in sorted order by number of classes, except that the GUI applications are grouped together at the bottom of the list.

Test case Adol-C is a package for automatic differentiation of algorithms [1] and COS is our *Class Ordering System* described in this paper. The ep matrix test case is an extended precision matrix application that uses *NTL*, a high performance portable C++ number theory library [23]. vkey is a GUI application that uses the *V GUI* library [29], a multi-platform C++ graph framework for GUI applications. The final three test cases are GUI applications: IV Edraw, IV Graphdraw and IV Drawserv were generated from the *IV Tools* drawing application [28], a suite of free XWindows drawing editors for Postscript, TeX and web graphics production.

The first column of data in Figure 8 lists the lines of code, LOC, not counting blank lines or comments. The second column lists the number of classes in the test case and the third column lists the number of template or nested classes. The test case for our *Class Ordering System*, COS, contains 11 template classes from the standard C++ library [12] and there was a nested class in each of the IV Graphdraw and IV Drawserv test cases.

6.2. Cycles in the ORDs

Figure 9 lists summary information about the ORDs and the cycles in the ORDs for the test cases in our study. The first two columns of data list information about the respective ORD and the final four columns list information about the strong components, SCCs, constructed during *Step 2* of the algorithm of Figure 6.

The first column of data in Figure 9 lists the number of classes and the second column lists the number of edges in the ORD for the respective test case. The third column lists the number of SCCs that contain a single class and the fourth column lists the number of SCCs that contain more than one class. We use the cost model, in *Step 3* of our approach, to remove edges until all SCCs consist of a single class. The fifth column lists the number of classes and the final column lists the number of edges in the largest SCC.

To illustrate the significance of the data in Figure 9, the second row of data lists information about COS, our *Class Ordering System*, whose ORD consists of 38 classes and 128 edges. The ORD for the COS contains a cycle of three nodes consisting of classes Graph, Node and Cluster, illustrated in Figure 7. Cycles are an undesirable feature of an ORD or class diagram [26] and we were dismayed to discover this cycle. We intend to refactor the cycle out of our COS; however, some design patterns, such as the Visitor pattern [9], are cyclic by nature. There are 11 edges in the SCC of the COS with a cycle, as illustrated in row two, the sixth and final column of data in Figure 9.

The final row of Figure 9 lists information for the IV Drawserv test case. The fifth and sixth columns of the final row show that the ORD for IV Drawserv contains 110 classes and 4,722 edges. This is the largest SCC with a cycle in any of the test cases in our suite.

6.3. Edge types in cycles

Figure 10 lists the number of classes and the number of each type of edge in the largest SCC for test cases COS, IV Edraw and IV Drawserv. We use this information to explain some of our results about breaking cycles

Test case	**Classes**	**Total edges**	**SCCs: 1 class**	**SCCs: > 1 class**	**Classes in largest SCC**	**Edges in largest SCC**
Adol-C	16	111	7	2	5	63
Class Ordering System	38	128	35	1	3	11
ep matrix	50	164	50	0	1	4
vkey	46	226	27	1	19	143
IV Edraw	44	252	30	2	12	117
IV Graphdraw	151	1340	106	3	39	673
IV Drawserv	236	6460	118	3	110	4722

Figure 9. *SCCs with cycles*. **This table shows information about the number of SCCs with a single class, the number of SCCs with a cycle, and the number of nodes and edges in the largest SCC.**

Number of:	COS	IV Edraw	IV Drawserv
Classes	3	12	110
Inheritance edges	2	10	100
Association edges	0	2	51
Composition edges	0	0	0
Dependence edges	7	31	490
Polymorphic edges	2	74	4081
Owned Element edges	0	0	0
No. of Total edges	11	117	4722

Figure 10. *Nodes & edge types in largest cycle*. **This table shows the number of nodes (classes) and the types of edges in the largest cycle for three test programs.**

in the next section. To illustrate the data in the figure, consider column one, which lists 3 classes and 2 inheritance edges in the SCC with a cycle in COS, our *Class Ordering System*. This information can be verified by inspection of classes Graph, Node and Cluster at the top of Figure 7 in Section 5.3. In addition, the COS SCC with a cycle contained no association edges, no composition edges, 7 dependence edges and 2 polymorphic edges.

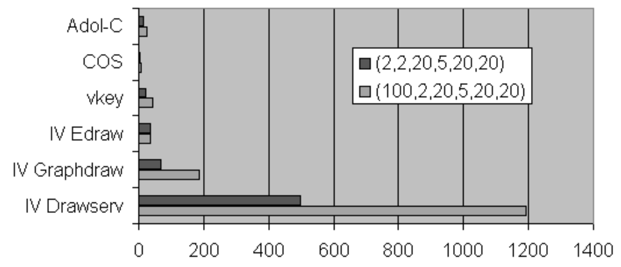
The third column of Figure 10 lists 4,081 polymorphic edges in IV Drawserv. This is almost forty times the number of classes in this test case. To see how the number of polymorphic edges can proliferate in an application with a preponderance of inheritance relationships, consider three classes in the IV Drawserv test case: base class OverlayCatalog, class FrameCatalog derived from OverlayCatalog, and class DrawCatalog derived from FrameCatalog. Class OverlayCatalog contains an association edge to itself, which generates 4 polymorphic edges from each of the base classes to each of the derived classes.

6.4. Breaking cycles in the ORD

Figure 11 lists results for the six test cases that contain a cycle larger than a single class. Each of the two columns of data in Figure 11 list the number of edges that were removed to break the cycles in the respective ORDs for two different cost models that varied by a single value. The cost model is described in Section 4.3. In the first column, we use a model that assigns weights of $(2, 2, 20, 5, 20, 20)$ to inheritance, association, composition, dependence, polymorphic and ownedElement edges respectively. These weight assignments are based on our estimation of the cost of stub construction for untested classes. The same weight assignments are used in the second column except that inheritance edges are assigned a weight of 100. We chose these weights based on an estimation of the cost of stub construction for untested classes (see Section 3). The bar graph in Figure 11 dramatizes the difference in the number of edges removed, especially for the IV Drawserv test case. The vertical axis of the bar graph lists the test cases and the horizontal axis shows the number of edges removed for each of the cost models.

Inheritance edges are considered poor choices for breaking cycles since they are likely to increase the complexity of stub construction [6, 7, 26]. Nevertheless, we chose to assign a low weight to the inheritance edge in the first model and a high weight to inheritance in the second model to investigate the impact on efficiency in breaking cycles. The low weight of 2 increases the probability that our heuristic will choose to remove an inheritance edge and the high weight of 100 makes it unlikely that an inheritance edge will be chosen to break a cycle.

Our results in Figure 11 indicate that including inheritance edges in edge removal considerations is more efficient for breaking cycles. For example, for Adol-C,



Test case	#edges removed **(2,2,20,5,20,20)**	#edges removed **(100,2,20,5,20,20)**
Adol-C	13	26
COS	2	6
vkey	20	44
IV Edraw	35	37
IV graphdraw	69	185
IV drawserv	498	1191

Figure 11. *Number of edges removed*. **The table lists the number of edges removed to break cycles using two cost models that differ by the weight assigned to inheritance edges. The bar graph highlights the difference in the two models.**

listed in the first row of the table, 13 edges were removed to break cycles for a low weight for inheritance and 26 edges were removed to break cycles for a high weight for inheritance. In fact, if inheritance edges are not removed, twice as many edge removals were required to break cycles in the SCCs for the test cases, with the exception of the IV Edraw test case where 35 and 37 edges were removed with and without inheritance edge removal respectively. The IV Edraw test case contains only 10 inheritance edges in its largest SCC, as shown on column two, row two of Figure 10. To break all of the cycles in the IV Edraw test case, 35 edge removals were required. With only 10 inheritance edges, the COS algorithm exhausted the supply of inheritance edges to remove before all cycles were broken; thus, the impact of the weight of the inheritance edge was mitigated by the scarcity of such edges in the IV Edraw test case.

The difference in the number of edge removals required for low and high weights for inheritance edges was more than double for the IV Drawserv test case, where 498 edges were removed to break cycles for the low inheritance weight and 1,191 edges were removed to break cycles for high inheritance weight. The difference in time to break cycles for the low weight was 20 seconds as compared to 98 seconds for the high inheritance weight. Considering the significant effort required to build stubs, the difference between 20 seconds and 98 seconds is not an imposing amount of time. On the other hand, the removal of 1,191 edges may require stubs for each of the respective supplier classes involved in the edge relationship. It has been conjectured that the number of stubs is proportional to the number of client classes that use the supplier classes, rather than the number of supplier classes [7]. Thus, an investigation into the complexity and number of stubs required for various types of removed edges is an important feature of our ongoing work.

7. Related Work

The focus of our work is a *Class Ordering System*, COS, that includes a parameterized cost model that permits flexibility in the types of edges chosen to break cycles in an ORD for a program under test. Our COS can accomodate C++ programs that contain template classes and functions and nested classes. In our survey of the literature we found no references that describe techniques to accomodate template classes or functions, or nested classes. Moreover, all references in the literature propose to break cycles using association edges, except for [16] where arbitrary edge types are removed. None of the references investigate the effect of choosing an alternative edge type to association to break cycles.

We reviewed references [6] and [14] in Section 3. We now review several other important works that relate to our paper.

Tai and Daniels use an ORD to generate a class testing order based on the number of incoming and outgoing edges [25]. To break cycles, edges are weighted by summing the number of incoming edges of the source node and the number of outgoing edges of the destination node. The edges with the higher weights are removed.

Briand et al. [7] propose a strategy for ordering classes for testing that combines two other approaches. A weight is computed by multiplying the number of incoming and outgoing edges for the vertices involved in each association edge in a strong component. Briand et al. conjecture that by selecting association edges that break the largest number of cycles, the number of stubs is minimized.

8. Conclusions

We have described our *Class Ordering System* that generates an order for the classes in a C++ application for class-based testing. Our results indicate that ORDs for C++ applications can contain cycles that include hundreds of classes and thousands of edges. We have shown that removing inheritance edges can increase the efficiency of the cycle breaking process.

However, previous research has conjectured that inheritance edge removal is likely to increase the complexity of stub construction [6, 7, 26]. Nevertheless, by refusing to remove inheritance edges, the cycle breaking algorithm may require more than twice as many edge removals, which will require stubs for the respective supplier classes. It has also been conjectured that the number of required stubs is proportional to the number of client classes that use the supplier classes rather than the number of supplier classes [7]. Thus, further investigation into the complexity and number of stubs that are required for various types of edge removals is an important feature of our ongoing work.

References

[1] A. Griewank and O. Vogel. http://www.math.tu-dresden.de/wir/project/adolc/, April 2003.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, second edition, 1974.

[3] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, second edition, 1990.

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

[5] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, 1999.

[6] L. Briand, J. Feng, and Y. Labiche. Using genetic algorithms and coupling measures to devise optimal integration test orders. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, pages 43–50, Ischia, Italy, July 2002. IEEE Computer Society Press.

[7] L. Briand, Y. Labiche, and Y. Wang. Revisiting strategies for ordering class integration testing in the presence of dependency cycles. In *Proceedings of the 12th International Symposium on Reliability Engineering (ISSRE '01)*, pages 287–297. IEEE Computer Society Press, Nov. 2001.

[8] M. Fowler and K. Scott. *UML Distilled*. Addison-Wesley, second edition, 1999.

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

[10] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman and Company, first edition, 1979.

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

[12] ISO/IEC JTC 1. *International Standard: Programming Languages - C++*. Number 14882:1998(E) in ASC X3. ANSI, first edition, September 1998.

[13] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*. Plenum Press, 1979.

[14] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. A test strategy for object-oriented programs. In *Proceedings of the 19th International Computer Software and Applications Conference (COMPSAC'95)*, pages 239–244. IEEE Computer Society Press, august 1995.

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

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

[17] B. A. Malloy, T. H. Gibbs, and J. F. Power. Decorating tokens to facilitate recognition of ambiguous language constructs. *Software, Practice & Experience*, 33(1):19–39, 2003.

[18] R. C. Martin. *Agile Software Development*. Prentice Hall, 2003. 0-13-597444-5.

[19] S. Matzko, P. Clarke, T. H. Gibbs, B. A. Malloy, and J. F. Power. Reveal: A tool to reverse engineer class diagrams. In *Proceedings of the International Conference on the Technology of Object-Oriented Languages and Systems*, Sydney, Australia, Feb 2002.

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

[21] OMG Unified Modeling Language Specification. http://www.omg.org/cgi-bin/doc?formal/03-03-01.pdf, March 2003.

[22] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, 1999.

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

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

[25] K.-C. Tai and F. J. Daniels. Test order for inter-class integration testing of object-oriented software. In *21st International Computer Software and Applications Conference, COMPSAC'97*, pages 602–607. IEEE, 1997.

[26] N. N. Thuy. Testability and unit tests in large object oriented software systems. In *Fifth International Software Quality Week*, San Francisco, CA, May 1992.

[27] D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2003.

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

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