Intra-Class Testing of Abstract Class Features

Peter J. Clarke, Djuradj Babich, Tariq M. King
School of Computing and Information Sciences
Florida International University
Miami, FL 33199, USA
{clarkep, dbabi001, tking003}@cis.fiu.edu

James F. Power
Department of Computer Science
National University of Ireland, Maynooth
Co. Kildare, Ireland
jpower@cs.nuim.ie

Abstract

One of the characteristics of the increasingly widespread use of object-oriented libraries and the resulting intensive use of inheritance is the proliferation of dependencies on abstract classes. Such classes defer the implementation of some features, and are typically used as a specification or design tool. However, since their features are not fully implemented, abstract classes cannot be instantiated, and thus pose challenges for execution-based testing strategies.

This paper presents a structured approach that supports the testing of features in abstract classes. Core to the approach is a series of static analysis steps that build a comprehensive view of the inter-class dependencies in the system under test. We then leveraged this information to define a test order for the methods in an abstract class that minimizes the number of stubs required during testing, and clearly identifies the required functionality of these stubs.

Our approach is based on a comprehensive taxonomy of object-oriented classes that provides a framework for our analysis. First we describe the algorithms to calculate the inter-class dependencies and the test-order that minimizes stub creation. Then we give an overview of our tool, AbstractTestJ *that implements our approach by generating a test order for the methods in an abstract Java class. Finally, we harness this tool to provide an analysis of 12 substantial Java applications that demonstrates both the feasibility of our approach and the importance of this technique.*

1. Introduction

The widespread use of object-oriented (OO) libraries, and more recently plug-ins for integrated development environments (IDEs), has led to an emphasis being placed on validating the correct behavior of these components. These components extensively use the properties of the OO paradigm, i.e., abstract data types, inheritance and dynamic binding of method calls. One of the properties of inheritance that increases its flexibility is the ability to defer the implementation of members in one class, an abstract class, to one or more of its concrete derived classes. Many of the previously mentioned components implement design patterns that use abstract classes as a key part of their implementation.

A plethora of testing techniques have been developed during the last two decades to test OO software. These OO testing techniques cover all the phases of the development cycle, from unit testing through system testing. Most of the testing techniques focus on aspects of the specification, implementation, or a combination of both. One area of testing OO software that has not received much attention is testing of abstract classes. This lack of concentration partially stems from the inability to instantiate objects of abstract classes, thereby preventing them from being able to be executed at runtime. However, there are benefits to be gained from testing the functionality of abstract classes. One such benefit is the reuse of test cases from the test history of the abstract class to validate any of its concrete derived classes [15]. In addition, testing the features of an abstract class to be used in libraries provides a level of confidence to the developer with respect to the correctness of the functionality derived from the abstract class. This testing activity can lead to the provision of higher quality libraries to be used by other software developers.

In this paper we present an approach to support the intra-class testing of the features in an abstract class. Our approach first classifies the characteristics of the classes in an OO application, then for each abstract class, identifies the methods of the abstract class that can be tested in its concrete descendants. The approach involves: performing lightweight dependency analysis of the inherited methods of the concrete descendants; generating an integration test order for the methods in the abstract class; and using the test order to minimize the number of stubs required when testing the inherited methods in the concrete descendants. To validate our approach and show its practicality, we have developed a testing tool *AbstractTestJ* and performed ex-

periments on a cross-section of OO applications. These applications range in size from a few hundred classes to over 21K classes.

The contributions of this paper are as follows:

1. The development of an approach to support the intra-class testing of methods in an abstract class through its concrete descendants by:

   (a) identifying *truly* inherited methods using dependency analysis, i.e., those methods with dependencies limited to the abstract class;

   (b) generating a test order that minimizes the number of stubs required when testing methods of an abstract class; and

   (c) minimizing the number of methods that need to be tested in a newly created concrete descendant class.

2. The development of a tool that fully implements and automates this approach.

This paper is organized as follows. Section 2 presents background information and Section 3 the motivation for the work. The overview of the testing approach is presented in Section 4. The detailed testing approach and tool support are described in Sections 5 and 6 respectively. Section 7 contains the experimental results. Section 8 presents the related work and we conclude in Section 9.

2. Background

This section provides background information on the notion of inheritance hierarchies and abstract classes, the current approaches to test the features of abstract classes, and how stubs are used during testing. We also introduce terminology that threads the paper.

2.1. Inheritance and Abstract Classes

Inheritance is one of the major concepts of the OO paradigm and can be used to provide extensibility and reusability of classes. Meyer [22] informally defines inheritance as a mechanism whereby a class is defined in references to others, adding all their features (fields and methods) to its own. This definition is captured formally as $R = P \oplus \Delta R$, where R denotes the newly defined class (or *child*), P the properties inherited from another class (or *parent*), and ΔR the new properties that differentiate R and P [26]. The operator $\oplus$ represents a method of combining the inherited and new properties.

The inheritance hierarchy may be viewed as a *tree* (single inheritance) due to the transitivity of the $\oplus$ operator, or as a *directed acyclic graph* (multiple inheritance) due to applying $\oplus$ operator to more than one parent. We use the term *ancestor* of a class C to refer to the immediate and non-immediate parents of a class. Similarly, the terms *descendant* refers to the immediate and non-immediate children of a class [26]. In this paper we focus on the semantics of inheritance as provided by the Java language [14].

An *abstract class* is a class that contains a *deferred* feature [22]. That is, at least one of the features in the interface of the class is not implemented. No instances of an abstract class can be created. A class inherited from an abstract class that is not abstract will be referred to as either a *concrete child* or a *concrete descendant* since we do not consider non-immediate descendants in this paper. Class features inherited unchanged are referred to as *inherited features* (*recursive* in [15]) and the overridden methods as *redefined* methods [15]. If an inherited method only has dependencies in the parent class where it is defined we refer to that method as a *truly* inherited method.

2.2. Testing Abstract Classes

Several approaches are described in the literature [5, 21, 28] that can be used to test the features of an abstract class. These are: (1) defining a concrete class solely for the purpose of testing the abstract class, (2) testing the abstract class as part of testing the first concrete descendant, (3) testing the minimal set of concrete descendants that do not redefine the methods in the abstract parent class, and (4) using guided inspection. The first three approaches are execution-based, similar to the approach we present in this paper. The test cases to validate the features in the concrete descendants can be generated using specification-based, implementation-based, or hybrid-based testing techniques [5, 15].

When testing the features of a concrete class there may be overhead incurred due to the creation of stubs. For example, a stub may be required if there is a cyclic dependency between two methods in a class or the there is a dependency to another method that is not available at the time of testing. Finding an integration test order for the features in a class is similar to finding a test order for a cluster of classes. That is, the test order is produced by generating a topological sort of the acyclic graph that represents the dependencies between the classes in the cluster. If there are cycles in the graph then one or more edges are removed, thereby requiring the use of stubs [6, 18, 19]. Note for our approach we assume all dependent methods are available at the time of testing.

3. Motivation

There has been little or no research presented in the literature on techniques to test the features of abstract classes.

App No.	Application/ Package Name	Domain	Ver.	Classes in App.	Abstract Classes	Concrete Children	Concrete Dependents	Impacted Classes
1	*SableCC* [13]	Parser Generator	3.2	281	32 (11%)	87	47	166 (59.1%)
2	*Soot* [24]	Java Optimization	2.2.2	2,230	184 (8%)	821	179	1,184 (53.1%)
3	*JRefactory* [2]	Reverse Eng. Tool	2.9.19	2,451	178 (7%)	807	209	1,194 (48.7%)
4	*AspectJ* [27]	Aspect-oriented Tool	9.1	2,933	213 (7%)	868	582	1,663 (56.7%)
5	*Twister* [12]	B2B oriented BPM	0.3	4,403	263 (6%)	775	535	1,573 (35.7%)
6	*Compiere* [1]	Integrated Framework	2.5.2	9,074	478 (5%)	2,485	779	3,742 (41.2%)
7	*org** [4]	BEA Web package	9.1	9,411	994 (11%)	2,061	1,005	4.060 (43.1%)
8	*JDK* [25]	Java Development Kit	1.5.05	15,418	1,187 (8%)	4,556	2,554	8,297 (53.8%)
9	*Netbeans* [23]	IDE	5.0	16,144	850 (5%)	3,056	1,575	5,481 (34.0%)
10	*Eclipse* [11]	IDE	3.1.1	17,794	965 (5%)	6,065	2,486	9,516 (53.5%)
11	*weblogic** [4]	BEA Web package	9.1	20,991	936 (4%)	5,533	1,871	8,340 (39.7%)
12	*com** [4]	BEA Web package	9.1	21,906	808 (4%)	6,061	1,556	8,425 (38.5%)
			Totals	**123,036**	**7,088 (6%)**	**33,175**	**13,378**	**53,641 (46.4%)**

Table 1. Summary of the Java applications used in the study. * indicates a package taken from a larger software application.

This dearth of testing techniques may have resulted from the fact that objects of abstract classes cannot be instantiated. However, abstract classes need to be adequately tested since they play an important role in object-oriented design. The *Stable Abstractions Principle* suggests that there should be a correlation between stability and abstraction [20]. Here, *stability* is a measure of the difficulty of changing a package or class, and is measured in terms of the number of other packages or classes that depend on it. Thus, a good object-oriented design will exhibit a high degree of dependencies on abstract classes. While this is advantageous, since abstract classes are easier to change than concrete ones, it means that if an abstract class should change, there is a high likelihood of a significant impact on the rest of the application.

To demonstrate this high level of dependency, we conducted a study of twelve (12) Java applications chosen from a variety of domains, ranging from compiler tools to application servers. Table 1 shows a summary of the applications used in our study. Columns 1 (*App No.*) through 4 (*Ver.*) of Table 1 contain the application numbers, name of applications, short descriptions and version numbers, respectively. The names in Column 2 of Table 1 postfixed with * indicate that the application used in the study is a package taken form a larger application.

The data in Columns 5 (*Classes in App.*) through 8 (*Concrete Dependents*) of Table 1 provide an estimate of the degree of dependence on abstract classes. Column 5 contains the total number of classes (excluding interfaces) cataloged and Column 6 shows the number abstract classes. As can be seen from Column 6, abstract classes constitute a relatively small percentage of the total number of classes, ranging from 4% to 11%.

The remaining columns in Table 1 measure the level of direct dependencies on these abstract classes. Column 7 shows the number of concrete children of abstract classes, and Column 8 shows the number of other classes that make a direct reference to a feature of an abstract class. Column 9 shows the total of Columns 6, 7 and 8, and provides an estimate of the number of classes that would be directly impacted by a change to an abstract class. As can be seen from comparing Columns 6 and 9, even though abstract classes form a relatively small percentage of the overall class count, any change to them would have a high impact, ranging from 35.7% to 59.1% of the classes in the applications studied. Note that this range is a conservative estimate, for a more accurate estimate additional analysis would have to be performed.

For example, application 12 in Table 1, *com**, is a package from the BEA Web Logic Server 9.1 [4] and version analyzed is 9.1. The application contains 21,906 classes of which 808 (4%) are abstract, 6,061 are the concrete children of the abstract classes, and 8,425 (38.5%) are directly impacted by the abstract classes.

4. Overview of Testing Approach

In this section we present an overview of the approach used to test the features of an abstract class by using its concrete descendants. The approach can be applied during unit and integration testing of a software application. We also present an illustrative example that threads the paper.

The steps of the testing approach are as follows:

Step 1: Catalog the classes in the component under test and identify the abstract classes (A_is) and concrete children ($C_{(A_i)j}$s) of each abstract class [3]. The intermediate data generated in the cataloging process will be required to support steps later in the approach.

Step 2: For each abstract class A_i:

(a) Perform dependency analysis on the set of methods in each $C_{(A_i)j}$.

(b) For each $C_{(A_i)j}$ identify the inherited, new, and redefined methods, using the data generated in Step 1.

(c) Generate a modified inter-method call graph for each $C_{(A_i)j}$ using the data in parts (a) and (b) to identify *truly* inherited methods, i.e. those methods that do not violate the dependency relationships in A_i .

(d) Find the near minimal set cover of the $C_{(A_i)j}s$ for A_i based on the truly inherited methods in the $C_{(A_i)j}s$.

(e) If the inherited features in the $C_{(A_i)j}$s do not cover the features in A_i, create a new concrete descendant $C_{(A_i)0}$ for the methods not in the partial set cover.

(f) Generate an inter-method call graph for the methods in A_i.

(g) Using the inter-method call graph, generate an integration test order for the methods in A_i that minimizes the number of stubs required.

(h) Generate a test order for the truly inherited methods in $C_{(A_i)j}$s, and if necessary those in $C_{(A_i)0}$, based on the test order generated in part (g).

(i) Using testing approaches described in Harrold et al. [15] and Binder [5], reuse test cases or create new test cases to test the inherited methods in A_i.

Illustrative Example. Figure 1 shows the Java code for the illustrative example that will be used throughout the paper. The code in Figure 1 shows a class hierarchy consisting of four classes: the abstract base class `ACsEx.A` (A_i), as well as classes `ACsEx.B1, ACsEx.B2` and `ACsEx1.B3`, the three concrete descendant of `A`, the $C_{(A_i)j}s$. There is also class `ACsEx.P` used by the methods in classes `ACsEx.A`, `ACsEx.B1` and `ACsEx1.B3`.

The example was constructed to include several of the features typical in large Java applications, including: inheritance of features across packages; inherited fields and methods; redefined methods; a combination of access specifiers for the methods in the abstract class `ACsEx.A`, including features declared as *package private*; and various combinations of bindings of fields to methods.

```
1  //Public class P in package ACsEx
2  package ACsEx;
3  class P{
4   protected int x;
5  }
6  // Public class A in package ACsEx
7  public abstract class A{
8    int x, y; P p1;
9    private int z;
10   protected abstract void a0();
11   void a1(P inP1){x = inP1.x; a4(p1);}
12   void a2(){y = x*x; a8(y);}
13   void a3(){y = y*p1.x;}
14   public void a4(P p2){p1.x = p2.x;
15     y = y*y; a6();}
16   private void a5(){a1(p1);}
17   public void a6(){a1(p1); a5();}
18   public void a7(){a5(); a2();}
19   public void a8(int i){
20     if(i != 0){x = x*10;
21     a8(--i);}}
22 }
23 // Classes B1 and B2 in package ACsEx
24 class B1 extends A{
25   int x;
26   public B1(int inX, int inY, P inP1){
27     x = inX; y = inY; p1 = inP1;}
28   protected void a0(){}
29   void a1(P inP1){x = x+inP1.x;}
30   void a5(){x = x+1;}
31   public  void a6(){a5(); a2();}
32   public void a8(int i){
33     if(i != 0){x = x*100;
34     a8(--i);}}
35   void b1(int x){this.x = x;}
36 }
37 class B2 extends A{
38   protected void a0(){}
39   void a2(){super.a2();}
40   void a3(){super.a3();}
41 }
42 // Class B3 in package ACsEx1
43 package ACsEx1;
44 import ACsEx.*;
45 class B3 extends A{
46   int x, y; P p1;
47   public B3(){}
48   public B3(int inX, int inY, P inP1){
49     x = inX; y = inY; p1 = inP1;}
50   public void a0(){x = p1.x;}
51 }
```

Figure 1. Example source code.

5. Testing Approach

Our testing approach consists of four major parts which correspond to the steps described in Section 4: (1) cataloging the classes in the component under test resulting in the identification of the abstract classes and their respective concrete descendants, and the classification of the features in these classes - Step 1; (2) identifying the truly inherited methods in the concrete descendants of each abstract class by performing lightweight dependency analysis - Step 2(a)-(c); (3) generating a near-minimal set cover of the concrete descendants for each abstract class using the truly inherited methods - Step 2(d)-(e); and (4) generating an integration test order for the methods in each abstract class to minimize

Classification	Entity list
Abstract Parent	`ACsEx.A`
Concrete Children	`ACsEx.B1`, `ACsEx.B2`, `ACsEx1.B3`
Class `ACsEx.B1`:	
Inherited Methods	`B1.a2()`, `B1.a3()`, `B1.a4(P)`, `B1.a7()`
Class `ACsEx.B2`:	
Inherited Methods	`B2.a1(P)`, `B2.a4(P)`, `B2.a6()`, `B2.a7()`, `B2.a8(int)`
Class `ACsEx1.B3`:	
Inherited Methods	`B3.a4(P)`, `B3.a6()`, `B3.a7()`, `B3.a8(int)`

Table 2. Shows the abstract class, the concrete descendants and their inherited methods cataloged from the code in Figure 1.

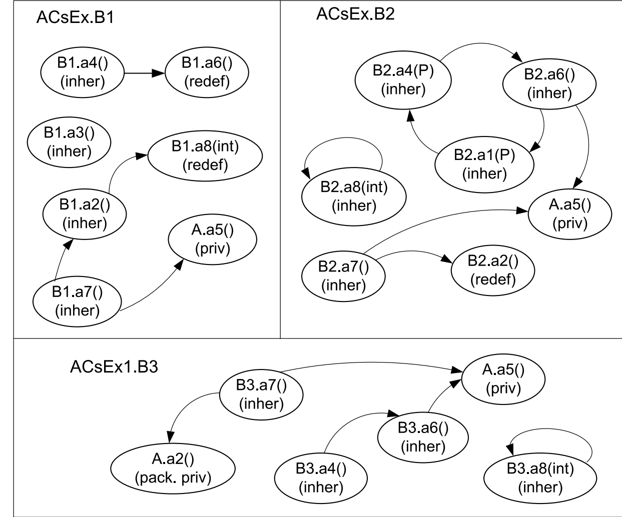

Figure 2. The modified call graphs for the classes `ACsEx.B1`, `ACsEx.B2`, and `ACsEx1.B3` shown in Figure 1.

the number of stubs required during testing - Step 2(f)-(h). We do not address how the actual test cases for the abstract classes are generated in this paper.

5.1. Feature Classification

The first step in our testing approach uses the Taxonomy of OO Classes by Clarke et al. [7, 8] to summarize the properties of the features in the abstract class and its concrete descendants. The taxonomy of OO classes facilitates the classification of an OO class *C* into a class group, and its features (fields and methods) into a set of feature groups. The classification is based on the characteristics exhibited by the class and its features. We define the *characteristics of a class* as the properties of the features in *C* and the dependencies *C* has with other types (built-in and user-defined) in the implementation. The properties of the features in *C* describe how criteria such as types, accessibility, shared class features, polymorphism, dynamic binding, deferred features, exception handling, and concurrency are represented in the fields and methods of *C*. The dependencies *C* has with other types are realized through declarations and definitions of *C*'s features, and*C's* role in an inheritance hierarchy [8]. The cataloged summary of a class and its features is contained in a *cataloged entry*.

In this paper we focus on the descriptors that indicate whether or not a class is abstract, and whether or not the methods of the derived class are inherited methods. Using the cataloged entries for the classes `ACsEx.A`, `ACsEx.B1`, `ACsEx.B2` and `ACsEx1.B3` from the code in Figure 1, the data in Table 2 was generated. Table 2 shows the abstract class, concrete descendants and their inherited methods.

5.2. Identifying Truly Inherited Methods

To determine if the behavior of an inherited method in the concrete descendant is similar to that of the defined method in the abstract superclass, we create a *modified inter-method call graph*. The modified inter-method call graph is created using the results of a lightweight dependency analysis as described in Section 5.4. To obtain the modified call graph we place two restrictions on the normal call graph. The first restriction is that each source vertex (a vertex with no incoming edges) in the graph represents an inherited method. Edges from the source vertex may go to any other vertex - representing methods accessible in the scope of the concrete descendant class. The second restriction is that a vertex representing a new or redefined method must be a sink (a vertex with no outgoing edges).

Figure 2 shows the modified call graphs for the concrete descendants `ACsEx.B1`, `ACsEx.B2`, and `ACsEx1.B3` whose source is presented in Figure 1. Each vertex in the call graph shown in Figure 2 is annotated with the name of the method and a property indicating whether the method is new, inherited (`inher`) or redefined (`redef`). We also annotate the methods that are private (`priv`) in `ACsEx.A`.

Using the modified call graphs for the concrete descendant classes, we determine if the inherited methods are truly inherited with respect to their behavior. This is done by determining the reachability of each vertex that represents an inherited method in the graph, where the relation is "calls". If the set of reachable methods from an inherited method m contains a vertex that represents either a new or rede-

fined method, then m is classified as not being truly inherited. For example, in Figure 2 the graph for the class ACsEx.B1 shows that the method B1.a7() is not truly inherited, since B1.a8(int) (redefined) is in the reachable set of B1.a7().

The inherited methods associated with the concrete classes, as shown in Table 2, can now be updated to reflect the true inherited methods. The only truly inherited methods are as follows: class ACsEx.B1 - B1.a3(); class ACsEx.B2 - B2.a1(P), B2.a4(P), B2.a6(), and B2.a8(int); and class ACsEx1.B3 - B3.a4(P), B3.a6(), B3.a7(), and B3.a8(int).

5.3. Minimal Set Cover

To obtain a near-minimal set cover of the truly inherited methods we use the greedy algorithm described in Cormen et al. [9]. The truly inherited methods in each concrete descendant $C_{(Ai)j}$ of the abstract class A_i are treated as a subset of X, where X is a finite set containing the implemented methods (methods that are not deferred) in the abstract class A_i. The concrete descendant classes are considered as a family $\mathcal{F}$ of subsets of X. The objective of the algorithm is to find a minimal set of subsets, $C_{(Ai)j}s$, in $\mathcal{F}$ that covers all the methods in X. Applying the algorithm to the example in Figure 1, we obtain the following partial set cover: class ACsEx.B1 - A.a3(); class ACsEx.B2 - A.a1(P), A.a4(P), A.a6(), and A.a8(int); and class ACsEx1.B3 - A.a7(). Note that we have to create a new concrete descendant class $C_{(Ai)0}$, referred to as A_IMPL, to test the methods A.a2() and A.a5() since these methods were not in the partial set cover.

5.4. Test Order to Minimize Stubs

The artifact used to generate the integration test order for the methods of the abstract class A_i is the *class inter-method call graph*. The approach we use is similar to the approach used to generate a class integration test order in [6]. However, unlike the approaches used to generate a class integration test order, the inter-method call graph has only one type of edge.

We use the results of a lightweight dependency analysis to generate the class inter-method call graph. The dependency analysis involves processing each class in turn, and examining the code in initializers and method bodies for references to features of other classes. A dependency graph is constructed whose nodes are the features of the class, and where directed edges represent the use of one feature by another. The call graph, as shown in Figure 3, is a strict sub-graph of this dependency graph.

Constructing the dependency graph is a two-pass operation. The first pass extracts the basic information from each class, while the second pass traverses the class hierarchy in order from parents to children. This second pass propagates the dependencies from parent classes to their children for each inherited method. At each propagation step, the dependencies on any dynamically-bound feature in the parent are updated to become dependencies on the corresponding feature in the children. Particular care must be taken with overridden attributes in Java, such as attribute x defined on lines 8 and 25 of Figure 1, to ensure that the bindings are maintained in a manner consistent with the rules in Section 8.3.3 of the Java Language Specification [14].

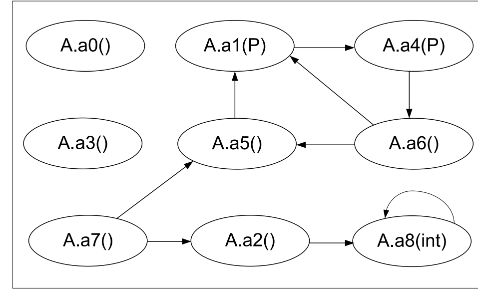

Figure 3. Call graph for the methods in the abstract class ACsEx.A shown in Figure 1.

Using the class inter-method call graph, we generate a test order to minimize the number of stubs required during testing as follows, using an approach similar to [6]. One possible test order for the call graph in Figure 3 is A.a4(P), A.a1(P), A.a8(int), A.a5(), A.a2(), A.a7(), A.a6(), A.a3(), A.a0(). One stub is required to simulate the behavior of A.a6() before A.a4(P) can be tested. Note that A.a8(int) makes a call to itself but we assume no stub is needed for methods that directly call themselves.

Using the test order stated in the previous paragraph and the set cover generated in Section 5.3, we can now generate a test order of the concrete descendant classes to minimize the use of stubs. The test order is as follows:

1. A.a4(P) tested in ACsEx.B2.
2. A.a1(P) tested in ACsEx.B2.
3. A.a8(int) tested in ACsEx.B2.
4. A.a5() tested in A_IMPL.
5. A.a2() tested in A_IMPL.
6. A.a7() tested in ACsEx1.B3.
7. A.a6(P) tested in ACsEx.B2.
8. A.a3() tested in ACsEx.B1.

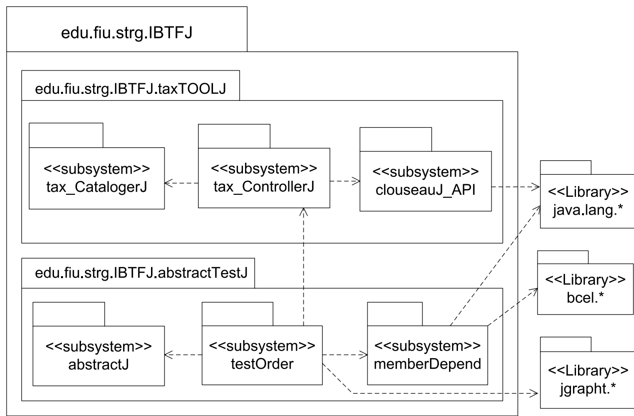

Figure 4. Package diagram for AbstractTestJ.

One stub is required when using the above test order i.e., a stub is required for `A.a6()` when testing `A.a4(P)`.

6. Tool Support

In this section we describe the tool used to support testing the features of an abstract class. The tool consists of two components: (1) *TaxTOOLJ - A Taxonomy Tool for the OO Language Java* [3] - that reverse engineers Java classes producing cataloged entries, and (2) *AbstractTestJ - An Abstract Testing tool for Java* - that generates a test order for the methods in an abstract Java class. These components are part of a larger testing framework - an *Implementation-Based Testing Framework for Java (IBTFJ)*, currently under construction. Figure 4 shows the package diagram for the tool containing TaxTOOLJ and AbstractTestJ. We refer to the test tool as *AbstractTestJ* in the remainder of the paper.

6.1. Cataloging Class Features

TaxTOOLJ catalogs the Java classes in a software application generating a cataloged entry for each class. The cataloging of classes is accomplished by utilizing the reflection facility provided by Java, and generating and inspecting the partial abstract syntax tree (AST) for each method in a class. TaxTOOLJ therefore provides the user with two options during the cataloging process: (1) *Reflection Only* and (2) *Complete Catalog*. In our experiments we only used the Reflection Only option.

The package `edu.fiu.strg.IBTFJ.taxTOOLJ` in Figure 4 shows the packages contained in TaxTOOLJ. These packages are: (1) clouseauJ_API - an interface that provides access to the details of the class to support the cataloging process, (2) tax_CatalogerJ - a repository that stores the cataloged entries, and (3) tax_ControllerJ - the subsystem that catalogs the classes in a Java application.

6.2. Generating the Test Order

The major subsystem responsible for generating the test order for the methods in the abstract class is the package `edu.fiu.strg.IBTFJ.abstractTestJ`, shown in the lower part of Figure 4. The package `abstractTestJ` consists of three subsystems: (1) `abstractJ` - stores the cataloged entries for the abstract classes and their concrete descendant classes, as well as the intra-class method call graph and other supporting graphs; (2) `testOrder` - implements the near-minimal greedy set-cover algorithm [9] and test ordering algorithm for the methods in the intra-class method call graph; and (3) `memberDepend` - performs a lightweight dependency analysis on the features of the abstract classes and their concrete descendants. This subsystem supports the construction of the intra-class method call graph (Figure 3) and modified method call graphs (Figure 2), respectively. All the graphs used in the tool were created using the *JGraphT* library [16].

The dependency analysis was performed using the BCEL [10] package, which provides an API allowing direct access to the bytecode stored in Java class files. Using BCEL at the class file level meant that even programs distributed without Java source code could be analyzed. Furthermore, the compiled bytecode in class files clearly and unambiguously identifies static and dynamic binding sites, as well as the fully-qualified attribute and method names, which greatly facilitates the accurate classification of the dependencies.

7. Experiments

In this section we present experiments that show the feasibility of applying our testing approach to the cross-section of Java 1.4.x and 1.5.x applications shown in Table 1. We first describe the experimental setup, then present our results, and conclude this section with a discussion of the results.

The experiments were performed on a Xeon 2.40 GHz PC with 3GBs of RAM. The settings for the JVM were -Xms1000m -Xmx1300m -XX:MaxPermSize = 256m, i.e., a minimum heap size of 1.0GB, a maximum heap size of 1.3GB and a maximum permanent generation size for the garbage collector at 256M. These settings were required due to the large number of classes that were loaded during analysis.

App. No.	App. Name	Entities Used During Testing: C_js/A_i	IMPLs/A_i	RMs/C_j	RMs/IMPL	Stubs/A_i	Time (sec) Catalog	Test Order
1	*SableCC*	1.00	0.06	7.13	2.00	0.00	2.2	3.3
2	*Soot*	0.76	0.39	8.07	5.79	0.29	10.0	27.0
3	*JRefactory*	0.78	0.63	10.17	5.81	0.12	10.8	39.8
4	*AspectJ*	0.79	0.59	13.43	6.05	0.51	21.7	72.6
5	*Twister*	0.64	0.74	9.15	9.72	0.22	33.6	43.3
6	*Compiere*	0.69	0.66	8.62	6.69	0.18	88.4	244.5
7	*org**	0.42	0.73	12.52	4.78	0.14	87.9	198.3
8	*JDK*	0.52	0.67	6.98	4.97	0.19	186.2	662.8
9	*Netbeans*	0.54	0.69	6.79	5.88	0.23	154.4	325.2
10	*Eclipse*	0.83	0.64	10.51	6.70	0.18	269.0	327.2
11	*weblogic**	0.68	0.67	8.48	6.35	0.20	252.8	361.1
12	*com**	0.61	0.73	12.55	6.28	0.28	315.7	428.2
	Summaries	**0.69**	**0.60**	**9.53**	**5.92**	**0.21**		

Table 3. Summary of the results obtained after applying our testing approach to the applications in Table 1. The following abbreviations are used in the column headings: A_i - abstract class, C_j - concrete direct descendant, RM - inherited method, IMPL - class created for testing purposes.

7.1. Results

Table 3 shows a summary of the results obtained when our testing approach is applied to the applications in Table 1. The first two columns in Table 3 are similar to their counterparts in Table 1. Columns 3 (C_js/A_is) through 7 (Stubs/A_i) show data obtained during the execution of our algorithms to generate the test order for the methods of the abstract classes in the applications. Column 3 contains the number of concrete direct descendants per abstract class (C_js/A_is) used during testing. Column 4 shows the data for the number of IMPL classes used per abstract class (IMPLs/A_is). Column 5 shows the number of methods from the abstract class that were tested in the concrete direct descendant on average (RMs/C_j). A similar number was generated for the IMPL classes in Column 6. Column 7 contains the number of method stubs per abstract class (Stubs/A_i) required during testing. We measured the average time (in seconds) required to perform the main parts of our algorithm as shown in Columns 8 and 9 of Table 3. Column 8 shows the average time to catalog the classes in the application and Column 9 the time to generate the test order for the abstract class methods being tested in the concrete direct descendants.

An example of the data presented in Table 3 is as follows: the *JDK* [25] application, shown in Row 8, used on average one concrete direct descendants class for every two abstract classes tested (Column 3 - 0.52), and used two IMPL classes for every three abstract classes tested (Column 4 - 0.67). Recall that the IMPL class is a newly created concrete child of the abstract class to be tested. Column 5 shows that on average approximately 7 (6.98) methods from the abstract classes tested in their concrete children and Column 6 approximately 5 (4.97) methods testing in the IMPL class. Column 7 shows that one method stub was required on average for every five abstract classes tested (0.19). Columns 8 and 9 show that to catalog the 15K classes in JDK took 186 seconds and to generate the test order for the methods in the abstract classes 662 seconds, respectively.

7.2. Discussion

In this section we highlight the characteristics of the results that substantiate the use of concrete descendant classes to test the features of abstract classes. We also consider other factors that need to be considered when using our approach, such as minimization of method stubs. In addition, we discuss threats to the validity of the results for our experimental data.

7.2.1 Approaches to Testing Abstract Classes

As mentioned earlier, the three dynamic approaches used to test the features of an abstract class include: (1) defining a concrete class solely for the purpose of testing the abstract class, (2) testing the abstract class as part of testing the first concrete descendant, and (3) testing the minimal set of concrete descendants that do not redefine the methods in the abstract parent class. The data in Table 3 favors our approach to testing the features of abstract classes. The summaries for Columns 5 and 6 in Table 3 indicate that on average 9.53 out of 15.45 (62%) of the methods in abstract classes

can be tested in their concrete direct descendants. If the first testing strategy (defining a concrete class for testing) is used, then the `IMPL` class for each abstract class would be created with 16 methods. After testing `IMPL` class the concrete direct descendants would still need to be tested. The one advantage of using the `IMPL` class to test the methods of the abstract class is that these methods are all tested in one class. That is, there is no need to interleave testing the methods in other classes due to the test order gnerated to minimize the number of stubs used, as in our approach.

The second approach to testing the features of an abstract class (as part of the first concrete descendant) is clearly not applicable in all situations. The example code in Figure 1 supports this claim, since several of the methods inherited by the classes `B1` and `B2` from the abstract class `A` are redefined. The results in Table 3 also support this claim due the number of `IMPL` classes required during testing. On average, for every five abstract classes tested, three `IMPL` classes are needed to test some of the methods in the abstract classes (Summaries Row of Column 4 Table 3). Using the third approach mentioned above without performing dependency analysis can result in the incorrect behavior of the methods in the abstract class being tested. The discourse presented in Subsection 5.3 supports this claim.

Our approach attempts to minimize the number of stubs required when testing the features of an abstract class (A_i) by generating an integration test order for the inherited methods in the concrete descendants ($C_{(A_i)j}$s). The order generated by our approach may not be optimal when considering the testing effort for each individual $C_{(A_i)j}$, since we did not consider the integration test order for all the methods in the $C_{(A_i)j}$. Our focus was on using truly inherited methods for testing A_i. Note that the truly inherited methods need not be retested when the concrete descendant classes are being tested after testing the abstract classes.

7.2.2 Threats to Validity

Since there are no other empirical studies in the research literature comparing the approaches to testing the features of abstract classes, it is difficult to completely validate our results. However, we partially validated the results by generating similar numbers in two components of the tool. For example, we generated the number of abstract classes and the number of concrete direct descendants in *TaxTOOlJ* and `memberDepend`, which uses the BCEL package. The results generated from both components were consistent. The subsystems of the `abstractTestJ` package (see Figure 4) were validated using the illustrative example that threads the paper and performing a structured walkthrough using the *SableCC* application shown in Table 1.

There are several limitations of the study including: (1) finding the class libraries required by the various applications, (2) using the library JGraphT [16], and (3) limitations of the JVM. During the preparation of the applications for the study, it was difficult to obtain all the libraries used by the applications being analyzed. The JGraphT library is a very useful library for generating and manipulating graphs. However, a few operations are based on pointer arithmetic, e.g., comparing if two vertices in a graph are equal. Several problems were encountered with the JVM when we attempted to catalog large applications. The main problem was an out of memory error. This was the main reason for splitting the BEA Web logic [4] into three packages.

8. Related Work

The research literature on testing the features of abstract classes is sparse. The work by Thuy [28] is most closely related to our work. Thuy presents three rules for testing abstract classes. These include: (1) deferred methods must be redefined in a concrete class and can be tested in that class, (2) an inherited method can be tested in the framework of a concrete derived class that does not redefine it, and (3) an inherited method in a concrete class that calls an extended method in a derived class must be tested in the derived class. The example presented in [28] uses the above rules to test the methods of the abstract class, and in some cases the same methods are tested in multiple concrete descendants. Thuy also suggests that it may be better, from a test management perspective, to test all methods of the abstract class in one concrete descendant if possible, or to create minimal set cover. However, this claim was not validated by any study to the best of our knowledge. We extended the rules presented by Thuy to include the test order of the concrete descendants in the minimal set cover and the use of a new concrete descendant if there is no set cover of the existing concrete descendants. In addition, we perform a lightweight dependency analysis to ensure that the behavior of the inherited methods tested in the concrete descendants is similar to that of the abstract base class.

Kong and Yin [17] describe new testing principles and an extension of the JUnit tool to support the testing of abstract classes. Their testing approach is based on the parallel architecture of class testing (PACT) [21] and uses a factory design model when used with JUnit to test abstract classes. For an abstract class A and concrete descendant B, *Tester* classes are created in JUnit. This is done to implement the PACT architecture. The A Tester class, which is also abstract, contains methods to test the features of A. The B Tester class inherits from the A Tester class and implements the abstract methods from A Tester. The B Tester implementation is then used to perform testing on A Tester, thereby testing A. Our approach does not provide the implementation details to perform the actual unit testing of the abstract class as in the work by Kong and Yin.

9. Concluding Remarks

In this paper we have presented an approach that supports the intra-class testing of the features in an abstract class. The approach generates a method integration test order for an abstract class that minimizes the number of method stubs required, and identifies the methods in the abstract class that can be tested through its concrete descendants. We also presented a description of the testing tool (*AbstractTestJ*) that implements our approach, and the results of experiments performed on classes from a cross-section of Java applications using AbstractTestJ. Our future work includes performing comparative studies with other techniques used to test the features of abstract classes.

10. Acknowledgments

This work was supported in part by the National Science Foundation under grant HRD-0317692.

References

[1] ComPiere, Inc. Compiere, August 2005. `http://www.compiere.org/`.

[2] M. Atkinson. JRefactory, May 2004. `http://jrefactory.sourceforge.net/`.

[3] D. Babich, K. Chiu, and P. J. Clarke. TaxTOOLJ: A tool to catalog Java classes. In *18th International Conference on Software Engineering and Knowledge Engineering*, pages 375 – 380, July 2006.

[4] BEA Systems, Inc. WebLogic Server 9.1 , Dec. 2005. `http://www.bea.com/`.

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

[6] L. Briand, Y. Labiche, and Y. Wang. Revisiting strategies for ordering class integration testing in the presence of dependency cycles. In *12th International Symposium on Software Reliability Engineering*, pages 287–297, Los Alamitos, CA, Nov. 2001.

[7] P. J. Clarke and B. A. Malloy. Using a taxonomy to analyze classes during implementation-based testing. In *8th IASTED Intl. Conf. on Software Engineering and Applications*, pages 288–293, Nov 2004.

[8] P. J. Clarke and B. A. Malloy. A taxonomy of OO classes to support the mapping of testing techniques to a class. *Journal of Object Technology*, 4(5):95–116, 2005.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT, and McGraw Hill, Cambridge, MA, 2nd edition, 2001.

[10] M. Dahm, J. van Zyl, E. Haase, D. Brosius, and T. Curdt. Byte Code Engineering Library v5.2, June 2006. `http://jakarta.apache.org/bcel/`.

[11] Eclipse Foundation. Eclipse, Jan. 2007. `http://www.eclipse.org/`.

[12] F. D. C. Fernandes and M. Riou. Twister, Mar 2005. `http://www.smartcomps.org/confluence/display/twister/Home`.

[13] E. M. Gagnon, B. Menking, M. Nowostawski, K. K. Agbakpem, and K. Gergely. SableCC, Dec 2005. `http://sablecc.org/`.

[14] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Prentice Hall, 3rd edition, 2005.

[15] M. J. Harrold, J. D. McGregor, and K. J. Fitzpatrick. Incremental testing of object-oriented class structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 68–80. ACM, 1992.

[16] JGraphT Development Team. JGraphT 0.7.0, July 2006. `http://jgrapht.sourceforge.net/`.

[17] L. Kong and Z. Yin. The extension of the unit testing tool Junit for special testings. In *1st Intl. Multi-Symposium on Computer and Computational Sciences, Vol 2*, pages 410–415, Washington, DC, 2006.

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

[19] B. A. Malloy, P. J. Clarke, and E. L. Lloyd. A parameterized cost model to order classes for class-based testing of C++ applications. In *14th International Symposium on Software Reliability Engineering*, pages 353–364. IEEE Computer Society, 2003.

[20] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2003.

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

[22] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, 1997.

[23] NetBeans Development Team. NetBeans, Jan. 2006. `http://www.netbeans.org/`.

[24] Soot Contributors. Soot, Aug 2005. `http://www.sable.mcgill.ca/soot/`.

[25] Sun Microsystems, Inc. Core Java J2SE 5.0, February 2005. `http://java.sun.com/j2se/1.5.0/`.

[26] A. Taivalsaari. On the notion of inheritance. *ACM Comput. Surv.*, 28(3):438–479, 1996.

[27] The AspectJ Team. AspectJ, Dec. 2005. `http://www.eclipse.org/aspectj/`.

[28] N. Thuy. Testability and unit tests in large object-oriented software. In *5th Intl. Software Quality Week*, pages 1–9. Software Research Institute, May 1992.