SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. (2010)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr.429

A testing strategy for abstract classes

Peter J. Clarke[1,*,†], James F. Power[2], Djuradj Babich[1] and Tariq M. King[3]

[1]*School of Computing and Information Sciences, Florida International University, 11200 S.W 8th Street, Miami, FL 33199, U.S.A.*
[2]*Department of Computer Science, National University of Ireland, Maynooth, Co. Kildare, Ireland*
[3]*Department of Computer Science, North Dakota State University, Fargo, ND 58108, U.S.A.*

SUMMARY

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. Since abstract classes cannot be instantiated, they cannot be tested in isolation using standard execution-based testing strategies. A standard approach to testing abstract classes is to instantiate a concrete descendant class and test the features that are inherited. This paper presents a structured approach that supports the testing of features in abstract classes, paying particular attention to ensuring that the features tested are those defined in the abstract class. Two empirical studies are performed on a suite of large Java programs and the results presented. The first study analyses the role of abstract classes from a testing perspective. The second study investigates the impact of the testing strategy on the programs in this suite to demonstrate its feasibility and to comment on the pragmatics of its use. Copyright © 2010 John Wiley & Sons, Ltd.

Received 2 December 2008; Revised 30 December 2009; Accepted 6 January 2010

KEY WORDS: abstract classes; object-oriented testing; test ordering

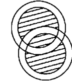
1. INTRODUCTION

The object-oriented (OO) software development paradigm places strong emphasis on code reuse, and a typical OO software application will make use of variety of OO libraries. These libraries range from language-standard library packages, through domain-specific application programming

*Correspondence to: Peter J. Clarke, School of Computing and Information Sciences, Florida International University, 11200 S.W 8th Street Miami, FL 33199, U.S.A.
†E-mail: clarkep@cis.fiu.edu

Contract/grant sponsor: National Science Foundation, U.S.A.; contract/grant numbers: HRD-0833093, HRD-0317692

Copyright © 2010 John Wiley & Sons, Ltd.



interfaces (APIs) to platform-specific libraries such as plug-in environments for integrated development environments (IDEs). Such a software makes extensive use of the key elements of the OO paradigm, namely abstract data types, interface and implementation 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 subclasses. Indeed, many modern software applications implement design patterns that use abstract classes as a key part of their implementation [1]. However, there are few empirical studies that report the impact that abstract classes have on concrete classes in these applications if changes are made to the abstract classes.

Testing OO software continues to be a major challenge due to several inherent properties of the OO design principles. These principles include encapsulation, generalization/specialization and information hiding which decrease the external observability of the state of objects at runtime [2]. Many of the techniques used to test OO software are based on techniques developed for pre-OO software. These testing techniques usually 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 executed at runtime. Ideally, this inability should not prevent the features of an abstract class from being adequately tested. For example, it should be possible to indirectly instantiate an abstract class via one of its concrete descendants and test the derived instance [2]. However, it is not clear if this strategy will fully test all the original features of the abstract class, since it is now effectively being tested as part of a system of classes. In traditional terms, the *antidecomposition* axiom defined by Weyuker states that if a component is tested it does not necessarily imply that the parts of the component have been adequately tested [3].

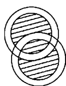
There are benefits to be gained from testing the functionality of abstract classes. One such benefit is the reuse of test cases from the test history of the abstract class to validate any of its concrete derived classes [4]. In addition, testing the features of an abstract class to be used in libraries 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.

This paper presents a strategy for testing abstract classes which is an extension of the work presented by Clarke *et al.* [5]. The underlying idea of the strategy is that since objects of an abstract class cannot be instantiated, the methods of the abstract class must be tested in a concrete subclass of the abstract class. However, a simplistic approach would have to insist that (1) the inherited methods cannot be overridden, and (2) no inherited method can depend on another method in the abstract class that is overridden. The strategy must therefore be able to identify those methods that satisfy both restrictions, referred to as *truly inherited methods* and those methods that do not.

The approach presented in this paper first classifies the characteristics of the classes in an OO application, and then identifies the methods of each abstract class that can be tested through its concrete descendants. This approach is founded on a lightweight dependency analysis of the inherited methods of the concrete descendants. The lightweight dependency analysis is used to generate an integration test order for the methods in the abstract class. Finally, this test order is used to minimize the number of stubs required when testing the inherited methods in the concrete descendants.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr



The major contributions of this paper include two empirical studies that were performed to further validate the strategy used to test abstract classes presented by Clarke *et al.* [5]. The applications used in the studies were the 16 large open-source Java applications from a corpus used by Melton and Tempero in their study on dependency and inheritance relationships between classes [6,7].

The first study shows that even though abstract classes form a relatively small proportion of the overall number of classes, a high proportion of concrete classes have a direct dependence on them. This study was motivated by the *Stable Abstractions Principle* (SAP) which suggests that there should be a correlation between stability and abstraction [8]. The second study investigates the pragmatics of testing the features of abstract classes. In particular, a comparison is made between the testing strategy presented in this paper and the other execution-based testing strategies in the literature. In addition, an analysis of the testing strategy is performed based on six different test ordering approaches that use different edge selection criteria for breaking the cycles in an inter-method dependency graph.

The remainder of this paper is organized as follows. Section 2 describes the foundational concepts and related work for testing abstract classes. Section 3 describes the first empirical study that analyses abstract classes with respect to their impact on concrete classes. Section 4 presents the strategy for testing abstract classes. Section 5 describes the pragmatics of the testing strategy which is based on a second empirical study that compares this strategy to other testing approaches for abstract classes. Finally, in Section 6 a review of some of the threats to validity for the empirical studies are described and the directions of future work are presented.

2. BACKGROUND AND RELATED WORK

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. The related work on testing the features of abstract classes is also presented in this section.

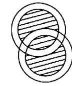
2.1. Inheritance and abstract classes

Inheritance is one of the major concepts of the OO paradigm and can be used to provide extensibility and reusability of classes. Meyer [9] informally defines inheritance as a mechanism whereby a class is defined in reference to others, adding all their features (fields and methods) to its own. The inheritance hierarchy may be viewed as a *tree* (single inheritance), or as a *directed acyclic graph* (multiple inheritance). The term *ancestor* of a class is used 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.

Meyer [9] defines an *abstract class* as a class that contains a *deferred* feature. That is, at least one of the features in the interface of the class is not implemented. The semantics for abstract classes usually do not allow instances of an abstract class to 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 non-immediate concrete descendants are not considered in this paper.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr



In the remainder of this paper the focus will be on the semantics of inheritance as provided by the Java programming language and described by Gosling *et al.* [10]. An abstract class in Java is one that either defines an abstract method, inherits an abstract method without defining it, or implements an interface without defining all its methods. Any attempt to directly create an instance of an abstract class in Java results in a compile-time error. An abstract *method* in Java is a method declared by signature only, with no method body.

In what follows inherited methods of a class are partitioned into two sets:

- Methods that are inherited unchanged are referred to as *inherited methods* (called *recursive* methods by Harrold *et al.* [4])
- Methods declared in a subclass with the same signature as methods declared in an ancestor and with a new implementation are referred to as *overridden methods* (called *redefined* methods by Harrold *et al.* [4])

If an inherited method only has dependencies in the parent class where it is defined that method is called a *truly inherited method*. This concept is discussed further in Section 4.3 and identifying truly inherited methods is a key aspect of the testing strategy described in the paper.

2.2. Testing abstract classes

The importance of devising specific approaches to testing the features of an abstract class has been discussed in some of the literature on software testing [2,11,12]. Each work notes that since abstract classes cannot be instantiated they cannot be directly tested. Overall, four general approaches are suggested to test the features of abstract classes:

1. Defining a concrete class solely for the purpose of testing the feature of the abstract class [11].
2. Testing the features of an abstract class as part of testing the first concrete descendant [11].
3. Testing a minimal set of concrete descendants that do not redefine the methods in the abstract parent class [12].
4. Performing a guided inspection on the features of the abstract class [11].

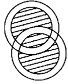
The first three approaches are execution-based, similar to the approach presented in the paper. Of the three execution-based approaches that are listed above, defining a concrete class for the purpose of testing the features of the abstract class is the only approach that can be used independently of the other two approaches stated. Approaches (2) and (3) may require the use of a newly defined concrete class to support testing. That is, if the first concrete class used to test the features of an abstract class redefines methods in the abstract class, or the set of concrete derived classes do not cover the set of inherited methods in the abstract class, then a new concrete class must be created solely for testing purposes.

The test cases to validate the features of the abstract class in its concrete descendants can be generated using specification-based, implementation-based, or hybrid-based testing techniques [2,4,11]. Binder defines several test patterns that can be used to test the features of concrete and abstract classes including *Polymorphic Server Test*, *Model Hierarchy Test*, *Abstract Class Test*, and *Generic Class Test* [2].

In addition, techniques that make use of the position of an abstract class (A) in the inheritance hierarchy can also be used. That is, if A is a derived class in an inheritance hierarchy then

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr



specification, implementation, and interface test cases can be reused for those methods that are inherited in A. These test cases are obtained from the test history of the parent of A and depending on whether the features in A are inherited or overridden [4]. Although the test cases for A may not be directly applied to the features of A they can however be used in one of the execution-based testing techniques stated at the beginning of this section.

2.3. Stubs

In the context of testing, a *stub* is defined as a skeletal or special-purpose implementation of a software module used to test another module that calls or is dependent on it [13]. The creation of stubs to support integration testing is considered to be a major cost factor, and therefore minimizing the number of stubs needed is a desirable goal of any testing approach [14–17]. The cost associated with the use of stubs derives from the need to develop and test their implementations [14].

Several integration testing techniques focus on generating an integration test order for a class cluster to reduce the number of stubs required during testing. Finding an integration test order for a cluster of classes is based on 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 [14–16,18,19]. In this paper the focus is on generating an integration test order for the methods in the abstract class and the stubs to support the testing of these methods.

There is relatively little work in the literature that discusses how to create method stubs, particularly stubs required when testing the methods of an abstract class. Ideally a stub implementation should be simple, so that there is little possibility of error, but should be capable of acting as a substitute for the original method during testing. McGregor and Skyes describe how to create stubs for the abstract methods inherited in a concrete class which has been created for the sole purpose of testing the abstract class [11]. The main idea is to create an implementation for each inherited abstract method in the descendant concrete class so that the implementation adheres to the postcondition of that abstract method. The use of mock objects can considerably ease the burden of generating implementations for classes during testing [20]. A tool such as EasyMock [21] facilitates the automatic creation of implementations of interfaces or extensions of classes during testing, and allows the user to add assertions regarding the number and order of method calls. In what follows the focus is on identifying the need for the creation of stubs, rather than the mechanism for creating them.

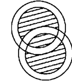
In addition to creating stubs for the abstract methods defined in an abstract class, there may also be a need to create stubs for some of the implemented methods in the abstract class. One or more stubs will be required if there are cycles in the call graph for the methods in the abstract class. The approach used in this paper is similar to the one described by McGregor and Sykes that creates a stub for the method that overrides the implemented method in the abstract class and adheres to its postcondition [11].

2.4. Related work

The research literature on testing the features of abstract classes is relatively sparse. The work by Thuy [12] is most closely related to the work presented in this paper. Thuy presents three rules for

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr



testing abstract classes:

- Deferred methods must be overridden in a concrete class and can be tested in that class.
- An inherited method can be tested in the framework of a concrete derived class that does not override it.
- 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 by Thuy 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 a minimal set cover. However, this claim appears not to be validated by any study in the research literature.

In this paper the rules presented by Thuy have been extended 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, a lightweight dependency analysis is performed to ensure that the behaviour of the inherited methods tested in the concrete descendants is similar to that of the abstract base class.

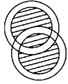
Kong and Yin describe new testing principles and an extension of the JUnit tool to support the testing of abstract classes [22]. Their testing approach is based on the parallel architecture of class testing (PACT) [11], and uses a factory design model with JUnit to test abstract classes. For an abstract class A and concrete descendant C, two *Tester* classes are created in JUnit in order to implement the PACT architecture. The A-Tester class, which is also abstract, contains methods to test the features of A. The C-Tester class inherits from the A-Tester class and implements the abstract methods from A-Tester. The C-Tester implementation is then used to perform testing on A-Tester, thereby testing A. The approach in this paper does not provide the implementation details to perform the actual unit testing of the abstract class as in the work by Kong and Yin.

There is a plethora of work on generating an integration test order for classes, a similar approach is used to generate the integration test order for methods in an abstract class [14–16,23]. The approach used in this paper is most closely related to the strategy proposed by Briand *et al.* [14] that merges the strategies by Le Traon *et al.* [16] and Tai and Daniels [23]. The approach by Briand *et al.* recursively identifies strongly connected components (SCCs) in an object relation diagram (ORD) using Tarjan's algorithm. The cycles in the non-trivial SCCs are broken based on a heuristic that uses the weights on the *association* edges. An association edge is one of the relationships used in UML class diagrams. This heuristic attempts to minimize the number of stubs required during testing. The approach in this paper uses a class inter-method call graph to represent the dependencies between the methods in an abstract class. Since the class inter-method call graph only contains one type of edge, cycles are broken using a heuristic that focuses on the in-degree and out-degree of the vertices in an SCC. In addition, the number of resulting SCCs after removing an edge in the SCC is also considered.

Hanh *et al.* [15] perform an experimental comparison of several techniques used to generate integration test orders based on the *cost* to create stubs and the number of steps needed to achieve integration, *duration*. The approach presented in the paper also indirectly measures the cost of developing a stub for a method. The cost is measured based on where the stub needs to be created. That is, if the stub for a method in a concrete class is not located in that class then it may be better

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr



to test the method in the concrete class created for testing purposes. The cost involved may not only be the creation of a method stub but also a stub for the concreted derived class.

The work presented in this paper is an extension of the previous work by Clarke *et al.* [5]. In this previous work an automated approach was presented to test the methods of an abstract class by testing the methods in the concrete descendant classes, or a new class created solely for the purposes of testing. The current paper extends this previous work in two main areas:

1. It describes an empirical study that shows the potential impact of an efficient testing strategy for abstract classes.
2. It presents a comparison of different test ordering strategies for abstract classes, focusing on the number of stubs required during testing.

When conducting the research presented in the paper by Clarke *et al.* [5], it became apparent that there was relatively little existing research on the role or importance of abstract classes in testing. In particular, given the difficulty of testing abstract classes, it is reasonable to ask whether the effort is worthwhile. In Section 3, a new analysis of 16 large Java applications which compares abstract and concrete classes from the perspective of testing is presented. This appears to be the first such study to concentrate on abstract classes, and to empirically verify their importance in testing.

3. AN EMPIRICAL STUDY OF THE USE OF ABSTRACT CLASSES

Given that there has been little or no research presented in the literature on techniques to test the features of abstract classes, it is reasonable to ask whether this is an important issue in software testing. In this section an analysis of abstract classes to motivate their importance in the testing process is presented. It is argued that abstract classes need to be adequately tested since they play an important role in OO design.

3.1. A suite of java applications

To investigate the importance of abstract classes in OO design a suite of open-source Java applications was selected and analysed. Originally a different suite of applications was selected based on a subjective evaluation of their merit. This initial suite of applications was similar to the one used by Clarke *et al.* [5]. However, a recent empirical study by Melton and Tempero used a large corpus of Java applications to study dependency and inheritance relationships between classes [6,7]. Since this study is relevant to the work presented in this paper, it was decided to base the study in this paper on the 16 largest (as measured by number of classes) open-source applications from their corpus.

One of the difficulties of comparing studies in empirical software engineering is that even when two studies agree on the programming language, they rarely agree on the applications, or the version of the applications selected. This compares poorly to other research areas such as performance evaluation where standardized benchmarks suites (such as the SPEC suite [24]) provide a common base of comparison for studies [25–27]. Hence, by using the same applications and versions as the study by Melton and Tempero progress can be made towards the re-use of existing empirical data.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

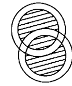


Table I. Summary of the 16 Java applications analysed in this paper. These are the 16 largest applications from the study of Melton and Tempero [6].

			Number of				
Application name	Version	Description	Class files	Inter-faces	Nested classes	Abstract classes	Concrete classes
ant	1.6.5	Java build tool	1014	60	314	52	588
argouml	0.18.1	UML drawing/critic	1393	98	142	77	1076
azureus	2.3.0.4	P2P filesharing	2953	476	1303	47	1127
columba	1.0	E-mail client	1335	115	155	37	1028
compiere	251e	ERP and CRM	1421	23	49	15	1334
derby	10.1.1.0	SQL database	1420	253	34	136	997
eclipse-SDK	3.1-win32	IDE	19 648	1830	8235	890	8693
geronimo	1.0-M5	J2EE server	2219	472	502	58	1187
hibernate	3.1-rc2	Persistence object mapper	1095	154	193	70	678
jasperreports	1.1.0	Reporting tool	736	110	103	43	480
jboss	4.0.3-SP1	J2EE server	5563	862	1414	184	3103
jre	14204	Java Runtime	10 303	1039	3047	782	5435
jtopen	4.9	iSeries & AS/400 Toolbox	3569	173	712	115	2569
netbeans	4.1	IDE	17 503	1191	9097	745	6470
pmd	3.4	Java code analyser	454	27	76	8	343
sandmark	3.4	Software watermarking	1100	37	251	81	731
		Totals	71 726	6920	25 627	3340	35 839

The 16 open-source Java applications used in the remainder of this paper are shown in Table I. The first three columns of Table I give the name, version number, and a brief description of each application. The remaining five columns give an approximate measure of the size of these applications. The total number of class files was used as the selection criterion for choosing the top 16 applications from the study of Melton and Tempero [6]. The remaining four columns break this down into interfaces, nested classes, concrete classes, and abstract classes. Since interfaces in Java contain no code they do not have a direct role in test-case ordering, and are not considered further in this study. The study by Melton and Tempero excluded nested classes, which are excluded in the remainder of the paper. While including nested classes is not necessary for any of the techniques described, excluding them at this point allows a comparison to be made with that of Melton and Tempero, and does not otherwise affect the findings presented in the paper.

3.2. Dependencies on abstract classes

As an initial attempt to gauge the importance of abstract classes in OO applications the proportion of abstract versus concrete classes was measured. Table II lists the 16 Java applications in the study, and the first and second column shows the application name and the total number of abstract and concrete (non-nested) classes. The third column shows the number of abstract classes and also expresses this as a percentage of the total number of classes. Initially these results were disappointing, since abstract classes form a relatively small proportion of the total number of classes in each application, ranging from 1.1% for compiere to 12.6% for jre.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

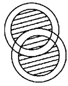


Table II. Summary of the impact of abstract classes in the 16 Java applications used in the study.

Application	# of Classes (excl. iface & nested)	Abstract classes (%)	Concrete children (%)	Concrete dependents (%)	Σ = all impacted (%)
ant	640	52 (=8.1%)	306 (=47.8%)	47 (=7.3%)	405 (=63.3)%)
argouml	1153	77 (=6.7%)	511 (=44.3%)	93 (=8.1%)	681 (=59.1%)
azureus	1174	47 (=4.0%)	141 (=12.0%)	55 (=4.7%)	243 (=20.7%)
columba	1065	37 (=3.5%)	287 (=26.9%)	54 (=5.1%)	378 (=35.5%)
compiere	1349	15 (=1.1%)	576 (=42.7%)	43 (=3.2%)	634 (=47.0%)
derby	1133	136 (=12.0%)	376 (=33.2%)	173 (=15.3%)	685 (=60.5%)
eclipse-SDK	9583	890 (=9.3%)	3760 (=39.2%)	1413 (=14.7%)	6063 (=63.3%)
geronimo	1245	58 (=4.7%)	162 (=13.0%)	65 (=5.2%)	285 (=22.9%)
hibernate	748	70 (=9.4%)	214 (=28.6%)	90 (=12.0%)	374 (=50.0%)
jasperreports	523	43 (=8.2%)	162 (=31.0%)	28 (=5.4%)	233 (=44.6%)
jboss	3287	184 (=5.6%)	582 (=17.7%)	247 (=7.5%)	1013 (=30.8%)
jre	6217	782 (=12.6%)	1957 (=31.5%)	1129 (=18.2%)	3868 (=62.2%)
jtopen	2684	115 (=4.3%)	843 (=31.4%)	400 (=14.9%)	1358 (=50.6%)
netbeans	7215	745 (=10.3%)	1972 (=27.3%)	1996 (=27.7%)	4713 (=65.3%)
pmd	351	8 (=2.3%)	78 (=22.2%)	2 (=0.6%)	88 (=25.1%)
sandmark	809	81 (=10.0%)	290 (=35.8%)	168 (=20.8%)	539 (=66.6%)
Summary:	39 176	3340 (=8.5%)	12 217 (=31.2%)	6003 (=15.3%)	21 560 (=55.0%)

Here the total number of classes includes abstract and concrete classes, but not interfaces or nested classes.

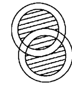
However, this small proportion does not give a full picture of the importance of abstract classes in an application. The *Stable Abstractions Principle* (SAP) suggests that there should be a correlation between stability and abstraction [8,28,29]. Here, *stability* is a measure of the difficulty of changing a package or class, and is measured in terms of the number of other packages or classes that depend on it. Thus, a good OO design ought to 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.

Motivated by the SAP, and making the assumption that the applications in the study represented ‘good’ OO design, the level of dependency was measured on abstract classes in each application, and these results are presented in the remaining columns of Table II. Column 4 shows the number and percentage of classes that are concrete children of abstract classes, and Column 5 shows the number and percentage of other classes that make a direct reference to a feature of an abstract class. Finally, Column 6 shows the total of Columns 3, 4, and 5 and thus provides an estimate of the number and percentage of classes that would be directly impacted by a change to an abstract class.

As can be seen from comparing Columns 3 and 6 of Table II, 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 at least 22.9% to as high as 66.6% of the classes in the applications studied. From this data it can be concluded that adequately testing abstract classes is an important issue to consider in testing an application.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr



3.3. Comparing abstract and concrete classes

While the study in the previous section exposes an important level of dependency on abstract classes, it does not address all the implications of the SAP. In particular, the SAP suggests that there may be an increased level of dependence on abstract classes when compared to concrete classes. This would suggest that not only are abstract classes important in the testing process but that they are, from one perspective at least, actually more important than concrete classes.

Melton and Tempero use two class-based metrics that are relevant to this question [6,30]:

- *fanin*, the number of Java source files that mention a class; this is similar to the calculation of dependencies in Table II above, but works at the file level rather than the class level.
- *scc*, the size of the (maximal) strongly connected component that the class is a member of. All classes in a single strongly connected component are cyclically dependent on one another.

Intuitively, the higher the *fanin* and *scc* values for a class, the greater the impact a change to that class will have on other classes in the application. To compare these values for abstract and concrete classes, two hypotheses are proposed:

$H_{\mathbf{fanin}}$**:** the *fanin* values for abstract classes are greater than those for concrete classes.
$H_{\mathbf{scc}}$**:** the *scc* values for abstract classes are greater than those for concrete classes.

To test these hypotheses the *fanin* and *scc* values calculated by Melton and Tempero[‡] are used. Since the metric values were not normally distributed a one-sided two-sample Wilcoxon test (also called a Wilcoxon–Mann–Whitney test) was used during the analysis.

The results of applying this test to the *fanin* and *scc* values are shown in Table III. For each application the attributes displayed are name, the mean value of the metric for abstract and concrete classes in the application, and the corresponding p-value associated with the Wilcoxon test. For each experiment, the null hypothesis is the opposite of H_{fanin} and H_{scc}, and thus values of $p<0.05$ allow these hypotheses with a 95% confidence interval to be accepted.

As can be seen from both sets of p-values in Table III, the H_{fanin} and H_{scc} at the 95% confidence level for all applications can be accepted, other than pmd. Thus, for most applications, the *fanin* and *scc* metrics are greater for abstract classes than for concrete classes. In the case of pmd it is notable that abstract classes form a particularly small proportion of the total number (just $3.4\% = \frac{9}{321}$), and over half of these abstract classes have *fanin* and *scc* values of 0. Further, nearly 39% ($= \frac{125}{321}$) of the classes are involved in one large strongly connected component which is chiefly (97%) composed of concrete classes.

3.4. Summary

In this section a study of 16 Java applications is presented to determine the importance that abstract classes might have in the testing process. It is shown that even though abstract classes form a relatively small proportion of the overall number of classes, a high proportion of concrete classes have a direct dependence on them. Further, the level of incoming dependence and the size of cyclic

[‡]Downloaded from http://www.cs.auckland.ac.nz/-/corpus.htm on 03/03/08.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

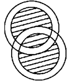


Table III. Results of using a Wilcoxon–Mann–Whitney test to test if the *fanin* and *scc* values for abstract classes are greater than those for concrete classes.

	fanin values			*scc* values		
Application	Abstract	Concrete	*p*-value	Abstract	Concrete	*p*-value
ant	8.0	2.6	<0.001	29.5	12.8	<0.001
argoUML	9.3	3.6	<0.001	223.7	187.2	0.023
azureus	8.7	1.9	<0.001	471.2	247.1	<0.001
columba	11.9	3.5	<0.001	4.5	2.8	<0.001
compiere	42.7	4.6	<0.001	30.5	15.6	0.0077
derby	8.3	4.1	<0.001	148.3	95.5	<0.001
eclipse	14.5	4.1	<0.001	101.7	66.8	<0.001
geronimo	8.8	1.3	<0.001	2.2	0.8	0.001
hiber	8.0	3.6	<0.001	454.7	324.5	<0.001
jasperreports	7.5	2.7	<0.001	88.5	47.6	<0.001
jboss	6.4	2.6	<0.001	17.8	12.3	<0.001
jre	12.4	5.9	<0.001	249.3	120.1	<0.001
jtopen	15.8	3.6	<0.001	74.0	30.1	<0.001
netbeans	6.2	1.5	<0.001	19.5	5.8	<0.001
pmd	12.6	2.3	0.34	27.3	29.3	0.66
sandmark	15.5	2.6	<0.001	8.5	4.5	<0.001

dependencies as measured by the *fanin* and *scc* metrics tend to be higher for abstract classes than for concrete classes in most applications.

4. A STRATEGY FOR TESTING ABSTRACT CLASSES

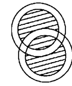
In this section an overview of the testing strategy is presented and a detailed description provided for each of the major parts of the algorithm for testing the features of abstract classes. The testing strategy consists of first classifying the features of abstract classes and then identifying truly inherited methods. Next a minimal set cover of the truly inherited methods is generated, and finally an integration test order for the abstract class methods is generated. In this paper the generation of actual test cases for abstract classes is not considered. An illustrative example that threads the paper is also presented in this section to elucidate the different parts of the testing strategy.

4.1. Overview

This subsection presents an outline of the algorithm that produces a testing strategy for the features in an abstract class. The input to the algorithm is the component (package(s) or class cluster(s)) containing the abstract classes and their immediate concrete descendants. The output for each abstract class consists of a test order for the methods in the abstract class, the list of stubs required during testing, the concrete derived classes where each method will be tested, and if needed a class developed solely to support testing.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr



The four main steps of the testing approach are as follows:

Step 1: *Catalog the classes in the component under test to identify class properties and features.*
This step identifies the abstract classes (A_is) and concrete children ($C_{(A_i)j}$s) of each abstract class [31]. The intermediate data generated in the cataloging process will be required to support steps later in the approach.

Step 2: *Identify the truly inherited methods in the concrete descendants of each abstract class.*
For each concrete child $C_{(A_i)j}$ of 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 2(a) and 2(b) to identify *truly* inherited methods.

Step 3: *Generate a near-minimal set cover of the concrete descendants for each abstract class using the truly inherited methods.*
For each abstract class A_i:

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

Step 4: *Generate an integration test order for the methods in each abstract class to minimize the number of stubs required during testing.*
For each abstract class A_i:

(a) Generate an inter-method call graph for the methods in A_i.
(b) Using the inter-method call graph, generate an integration test order for the methods in A_i that minimizes the number of stubs required.
(c) 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 4(b).
(d) Using testing approaches described in Harrold *et al.* [4] and Binder [2], reuse test cases or create new test cases to test the inherited methods in A_i.

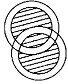
The following subsections elaborate on aspects of this algorithm using an illustrative example.

4.2. Feature classification

Figure 1 shows the Java code for the illustrative example that will be used throughout the paper. The corresponding class diagram for the Java code is shown in Figure 2. The code in Figure 1 contains 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 descendants of `A` (referred to as $C_{(A_i)j}s$ in the algorithms presented in the paper). There is also class `ACsEx.P` used by the methods in classes `ACsEx.A`, `ACsEx.B1`, and `ACsEx1.B3`. To aid in understanding the Java code, Figure 4 shows the call graphs for the methods in the classes `ACsEx.B1`, `ACsEx.B2`, and `ACsEx1.B3`.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr




```
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(); a8(x);}
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; a5(); a6();
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 showing four classes. Here `ACsEx.A` is an abstract class, and it has three concrete subclasses, `ACsEx.B1`, `ACsEx.B2` and `ACsEx1.B3`.

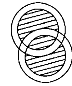
The example was constructed to include several of the features typical in large Java applications, including: inheritance of features across packages; inherited fields and methods; redefined methods; a combination of access specifiers for the methods in the abstract class `ACsEx.A`, including features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

Step 1 of the testing approach uses the Taxonomy of OO Classes by Clarke *et al.* [32,33] 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. The *characteristics of a class* are defined 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 [33]. The cataloged summary of a class and its features is contained in a *cataloged entry*.

In this paper the focus is on those 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

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr



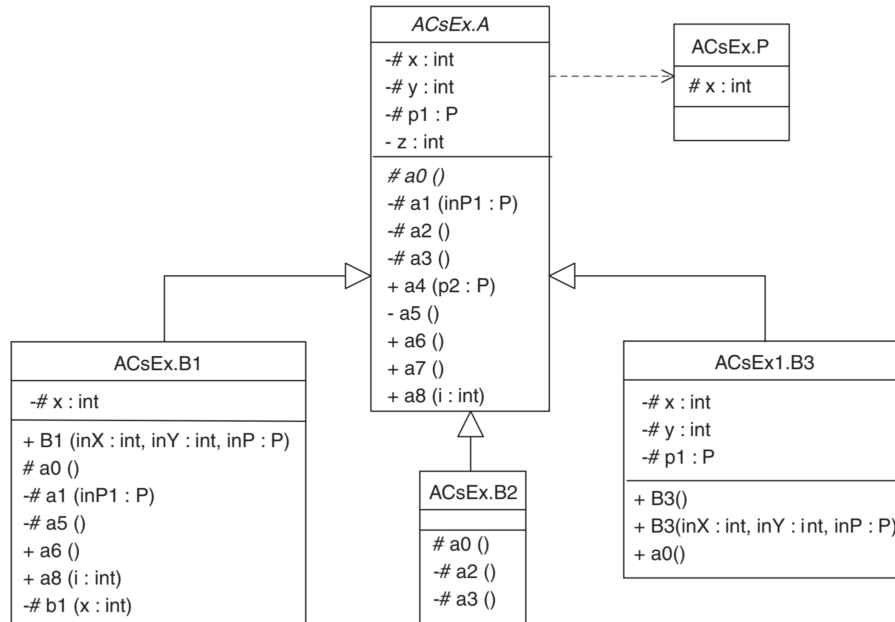


Figure 2. Class diagram for the example code shown in Figure 1. -# represents *package-private* accessibility.

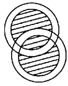
Table IV. The abstract class, the concrete descendants and their inherited methods cataloged from the code in Figure 1.

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)`

for the classes `ACsEx.A`, `ACsEx.B1`, `ACsEx.B2`, and `ACsEx1.B3` from the code in Figure 1, the data in Table IV was generated. Table IV shows the abstract class, concrete descendants, and their inherited methods.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr



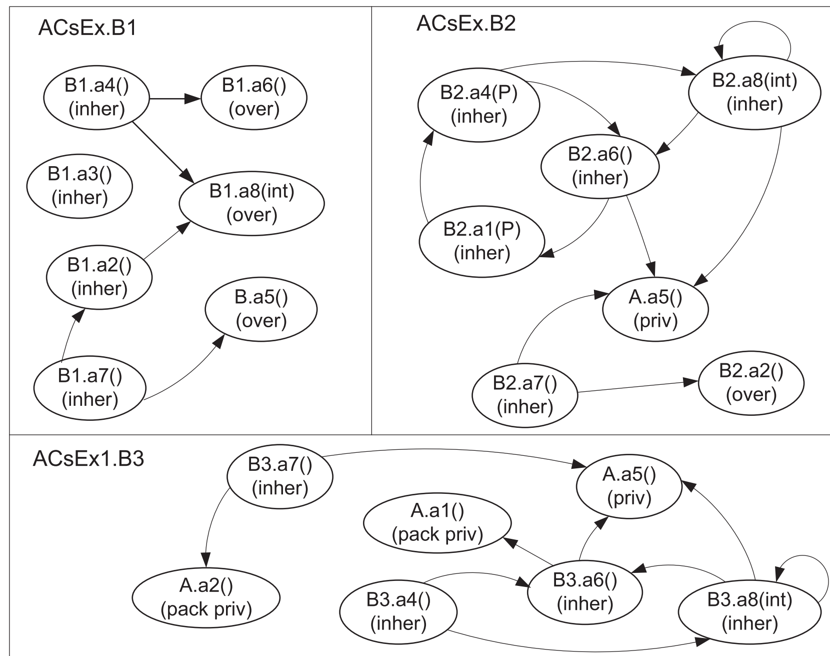


Figure 3. The modified call graphs for the classes `ACsEx.B1`, `ACsEx.B2`, and `ACsEx1.B3` shown in Figure 1.

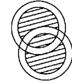
4.3. Identifying truly inherited methods

Step 2 of the testing approach involves identifying the *truly inherited* methods in a concrete descendant class. To determine if the behaviour of an inherited method in the concrete descendant is similar to that of the defined method in the abstract superclass a *modified inter-method call graph* is created. Identifying if the behaviour is similar for inherited methods requires that (1) the inherited methods cannot be overridden, and (2) no inherited method can depend on another method in the abstract class that is overridden. The modified inter-method call graph is created using the results of a lightweight dependency analysis as described in Section 4.5. To obtain the modified call graph two restrictions are placed 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 overridden method must be a sink (a vertex with no outgoing edges).

Figure 3 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 3 is annotated with the name of the method and a property indicating whether the method is new, inherited (`inher`), or overridden (`over`). The methods that are private in `ACsEx.A` are also annotated, (`priv`) for private to the class and (`pack priv`) for package private.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr



Using the modified call graphs for the concrete descendant classes, it is determined if the inherited methods are truly inherited with respect to their behaviour. 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 overridden method, then m is classified as not being truly inherited. For example, in Figure 3 the graph for the class `ACsEx.B1` shows that the method `B1.a7()` is not truly inherited, since `B.a5()` and `B1.a8(int)` are overridden and in the reachable set of `B1.a7()`.

The inherited methods associated with the concrete classes, as shown in Table IV, can now be updated to reflect the truly inherited methods. The only truly inherited methods in the three concrete descendant classes are as follows:

- In class `ACsEx.B1`: method `B1.a3()`.
- In class `ACsEx.B2`: methods `B2.a1(P)`, `B2.a4(P)`, `B2.a6()`, and `B2.a8(int)`.
- In class `ACsEx1.B3`: methods `B3.a4(P)`, `B3.a6()`, `B3.a7()`, and `B3.a8(int)`.

The methods that are not truly inherited are those methods in Table IV not stated in the itemized list above. For example, methods `a2()`, `a4(P)`, and `a7()` in `B1` are not truly inherited because `a2()` calls `a8(int)` which is overridden, `a4(P)` calls both `a6()` and `a8(int)` which are overridden and `a7()` calls `a5()` which is also overridden. If all the inherited methods were used during testing then when `a2()` is being tested the state of the variables `x` and `y` after executing `a2()` would be different in `B1` than if it were executed in `A`.

Since the approach presented in the paper does not take into account the semantics of how methods are overridden, it may be feasible to test `a2()` in `B2` since the approach used to override `a2()` does not affect the behaviour of `a2()`. That is, `a2()` is overridden in adherence to Liskov's substitution principle [34], i.e. `B2` is a subtype of `A`. While this makes the approach presented somewhat conservative in its analysis, it has the benefit of increased ease of implementation.

4.4. Minimal set cover

Step 3 of the testing approach involves generating a near-minimal set cover of the truly inherited methods. To obtain this near-minimal set cover the greedy algorithm described in Cormen *et al.* [35] is used.

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 $\mathscr{F}$ of subsets of X. The objective of the algorithm is to find a minimal set of subsets, $C_{(Ai)j}s$, in $\mathscr{F}$ that covers all the methods in X.

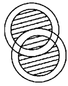
Applying the algorithm to the example in Figure 1 generates the following partial set cover:

- In class `ACsEx.B1`: method `A.a3()`.
- In class `ACsEx.B2`: methods `A.a1(P)`, `A.a4(P)`, `A.a6()`, and `A.a8(int)`.
- In class `ACsEx1.B3`: method `A.a7()`.

Note that a new concrete descendant class $C_{(A)0}$, referred to as `A_IMPL`, is created to test the methods `A.a2()` and `A.a5()` since these methods were not in the partial set cover.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr



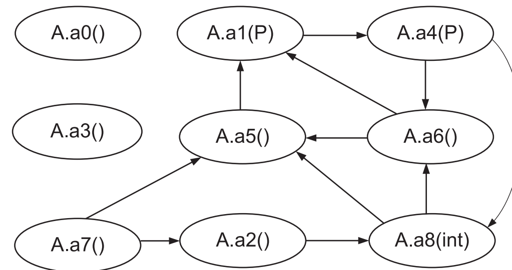


Figure 4. Call graph for the methods in the abstract class `ACsEx.A` shown in Figure 1.

4.5. Test order to minimize stubs

Step 4, the final step in the testing approach, involves generating an integration test order for the methods in each abstract class to minimize the number of stubs required during testing. The artifact used to generate the integration test order for the methods of an abstract class is the *class inter-method call graph*. The integration test order used is similar to the one described by Briand *et al.* [14] used to generate a class integration test order. However, unlike other approaches used to generate a class integration test order, the inter-method call graph has only one type of edge.

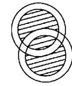
The results of a lightweight dependency analysis are used to generate the class inter-method call graph. The dependency analysis involves processing each class in turn, and examining the code in initializers and method bodies for references to features of other classes. A dependency graph is constructed 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 4, 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 [10].

The algorithm used to generate the test order for the methods in an abstract class is based on the algorithm presented by Kung *et al.* that orders the classes in a directed graph representing the dependencies among classes (called the *Object Relation Diagram*, or ORD) [19]. If there are cycles in the dependency graph Tarjan's algorithm is used to identify the SCCs [36]. For those SCCs with cardinality greater than one (referred to as a *super vertex*) an edge selection strategy is used to break the cycles in the SCCs to increase the number of SCCs. The edge selection criterion is chosen to minimize the number of edges to be removed in order to make the dependency graph acyclic.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr



Using the class inter-method call graph (*IMCG*), a test order is generated to minimize the number of stubs required during testing as follows:

1. Label any *SCC* with cardinality greater than one, $SCC_{|SCC|>1}$, as a *super vertex*.
2. Generate a reverse topological sort ordering (*RTS*).
3. For each $SCC_{|SCC|>1}$ in the *RTS*
   (a) Remove one or more edges to increase the number of SCCs based on an *edge selection criterion*.
   (b) For each edge that is removed a stub for the target vertex is required when testing the source vertex.
   (c) Perform Steps 1 through 3 until all SCCs in the RTS have a cardinality equal to one.

The class inter-method call graph shown in Figure 4 consists of three *SCC*s two with cardinality one and one with cardinality seven. By removing the edge (`A.a1(P)`, `A.a4()`) all remaining *SCC*s have cardinality equal to one. Here the edge selection criteria used is based on the maximum product of the in-degree of the source and out-degree of the target vertices [14]. After removing the edge (`A.a1(P)`, `A.a4()`) a method stub for `a4()` would be required when testing `a1(P)`.

One possible test order for the call graph in Figure 4 is `A.a1(P)`, `A.a5()`, `A.a6()`, `A.a8(int)`, `A.a2()`, `A.a7()`, `A.a4(P)`, `A.a3()`, `A.a0()`. One stub is required to simulate the behaviour of `A.a4()` before `A.a1(P)` can be tested. Note that `A.a8(int)` makes a call to itself but it is assumed that no stub is needed for methods that directly call themselves. Methods with recursive calls are validated using code inspection [11]. The results for three different *edge selection criteria* used to remove edges in an $SCC_{|SCC|>1}$ are presented in Section 5.4.

To generate a test order for methods in an abstract class using its concrete descendants the output of the test ordering algorithm, described in the previous paragraph, and the set cover of the concrete descendant classes generated using the approach described in Section 4.4 are combined.

Using the example in Section 4.1 one feasible test order for the truly inherited methods in the concrete descendant classes of `A` to minimize the use of stubs is as follows:

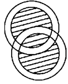
1. Method `A.a1(P)` is tested in class `ACsEx.B2`.
2. Method `A.a5()` is tested in class `A_IMPL`.
3. Method `A.a6(P)` is tested in class `ACsEx.B2`.
4. Method `A.a8(int)` is tested in class `ACsEx.B2`.
5. Method `A.a2()` is tested in class `A_IMPL`.
6. Method `A.a7()` is tested in class `ACsEx1.B3`.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

Recall that `A_IMPL` is a class created solely for the purpose of testing methods not covered in the concrete descendant classes. One stub is required when using the above test order i.e. a stub is required for `A.a4(P)` when testing `A.a1(P)`.

Figure 5 shows a dependency graph representing the test order shown above. The classes in which the methods will be tested are shown and the subscripts represent the test order. Since `A.a0()` is an abstract method in `ACsEx.A`, it will be tested in each of the concrete classes of `ACsEx.B2`.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr



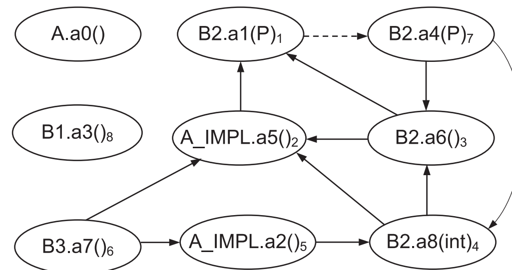


Figure 5. Dependency graph showing one test order of `ACsEX.A`'s methods and their concrete classes. The dashed line shows the edge that must be removed due to cycles in the dependency graph.

5. PRAGMATICS OF THE TESTING STRATEGY

In this section experiments are presented that show the feasibility of applying the testing approach to the suite of Java applications shown in Table I and previously discussed in Section 3. More specifically, this section contains an overview of the tool used to carry out the experiments, a brief description of the experimental environment for the results presented in this paper, and finally the results of using the testing strategy from two perspectives. The first perspective is a comparison with existing approaches to testing abstract classes, and the second perspective shows some of the pragmatics regarding the test ordering for the methods in abstract classes.

5.1. Tool support

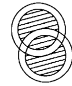
In order to validate the testing strategy described in Section 4 a tool was implemented that facilitated experimentation on the application suite described in Table I. The tool consists of two components: (1) *TaxTOOLJ—A Taxonomy Tool for the OO Language Java* [31]—that reverse engineers Java classes 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 6 shows the package diagram for the tool containing TaxTOOLJ and AbstractTestJ. The test tool is referred to as *AbstractTestJ* in the remainder of the paper.

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. The package `edu.fiu.strg.IBTFJ.taxTOOLJ` in Figure 6 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.

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 6. The package `abstractTestJ` consists of three subsystems: (1) `abstractJ`—stores

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr



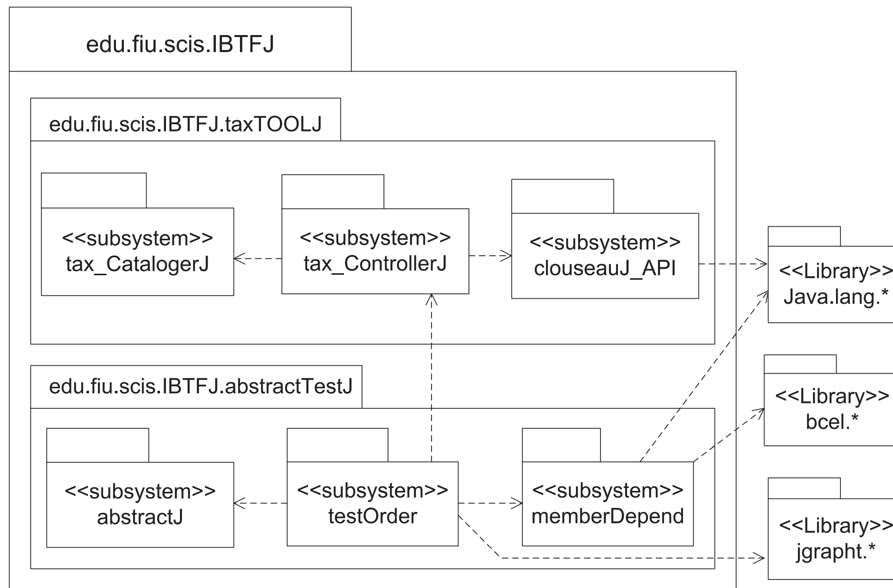


Figure 6. Package diagram for the tool used in this paper showing the elements of AbstractTestJ and TaxTOOLJ.

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 [35] 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 4) and modified method call graphs (Figure 3), respectively. All the graphs used in the tool were created using the *JGraphT* library [37].

The dependency analysis was performed using the BCEL [38] 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 analysed. 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.

5.2. Experimental environment

The experiments were performed on a Intel Core 2 Duo CPU @ 2.40 GHz with 3GBs of RAM. The settings for the JVM were $-\mathrm{Xms}1000\,\mathrm{m}$ $-\mathrm{Xmx}1300\,\mathrm{m}-\mathrm{XX}:\mathrm{MaxPermSize}=256\,\mathrm{m}$, i.e. a minimum heap size of 1.0 GB, a maximum heap size of 1.3 GB, and a maximum permanent generation size for the garbage collector at 256 M. These settings were required due to the large number of classes that were loaded during analysis. As stated in Section 3.1, 16 of the largest

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

(number of classes) open-source the applications used in the study were selected from the corpus used in a recent study by Melton and Tempero [6].

The preparation of the applications consisted of the following steps. The first step involved obtaining the `jar` files for each application and extracting the class files to the respective directories. The second step involved selecting the directory that corresponded to the package(s) used in the study by Melton and Tempero [6]. The third step involved removing the nested classes from the package(s) to be analysed thereby complying with the corpus described by Melton and Tempero [6]. The fourth and final step involved the creation of a repository to store all the libraries required by the applications in Table I. The libraries were required so that TaxTOOLJ would be able to analyse all the classes in the applications and not skip any classes due to the possibility of some classes not being loaded by the JVM.

5.3. Comparison with other testing approaches

In order to compare the approach to testing abstract classes, as described in Section 4, with existing approaches, a number of experiments was performed on the 16 Java applications previously listed in Table I. As mentioned in Section 2, the three main 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 purpose of the data in Table V is to quantify the advantages of the approach presented in the paper over the aforementioned three approaches.

Table V shows a summary of the results obtained when the testing approach is applied to the applications in Table I. The first column lists the name of each application, and the remaining four columns show data obtained during the execution of the algorithms to generate the test order for the methods of the abstract classes in the applications. It is important to note that these are the values for classes used during the testing of abstract classes, and not for the application as a whole. The time taken to generate the test order for the abstract class methods was in the range of 8.7 s for the jasperreports application and 702.4 s for the eclipse-SDK application.

Column 2 contains the average number of concrete direct descendants per abstract class (C_js$/A_i$s) used during testing. Column 3 shows the average number of synthetic `IMPL` classes created for each abstract class (`IMPL`s$/A_i$s). Since each `IMPL` class will test all the methods not tested by the concrete descendants, the average number of `IMPL` classes per abstract class will always be between 0 and 1. Column 4 shows the number of methods from the abstract class that were tested in the concrete direct descendant on average (Ms$/C_j$), while Column 5 shows the corresponding number for the `IMPL` classes. The last row shows the arithmetic averages of the contents of each column.

Examining the data in Column 2 of Table V shows that on average only 0.71 concrete direct descendants of abstract classes were used during testing, with a minimum of 0.38 for netbeans and a maximum of 0.88 for ant. It should be noted first that this figure is consistently less than the total possible number of concrete direct descendants per abstract class. From Table II it can be calculated that, on average, each abstract class has 3.66 concrete children. Thus it may be inferred that the selection of a minimum set cover yields considerable efficiency when compared to arbitrarily testing all concrete descendants. Using the raw data from Table I for ant it takes 46 concrete descendant

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

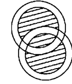


Table V. Summary of the results obtained after applying the testing approach to the applications in Table I.

	Entities used during testing			
Application	C_js/A_i	IMPLs/A_i	Ms/C_j	Ms/IMPL
ant	0.88	0.46	14.46	2.13
argouml	0.77	0.62	10.80	3.35
azureus	0.66	0.64	8.23	4.27
columba	0.84	0.54	7.68	4.00
compiere	0.80	1.00	16.08	3.40
derby	0.81	0.71	15.51	6.91
eclipse-SDK	0.79	0.72	11.26	6.25
geronimo	0.52	0.64	7.57	3.78
hibernate	0.87	0.51	15.75	3.39
jasperreports	0.65	0.42	29.86	2.94
jboss	0.74	0.62	3.30	12.20
jre	0.40	0.68	7.73	4.76
jtopen	0.86	0.61	13.34	4.13
netbeans	0.38	0.75	9.20	7.24
pmd	0.75	0.63	11.83	2.80
sandmark	0.60	0.57	11.14	3.98
Average	0.71	0.63	12.11	4.72

The following abbreviations are used in the column headings: A_i—abstract class, C_j—concrete direct descendant, M—inherited method, IMPL—class created for testing purposes.

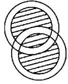
classes and 24 IMPL classes to test 52 abstract classes. The number of IMPL classes used during testing is independent of the number of concrete descendant classes, since the IMPL classes are required to test those methods that are not truly inherited in any of the concrete descendant classes. If the minimum set cover was not used in the testing strategy and it was decided to use all the concrete descendant classes then 76 classes (concrete descendants and IMPL) would be required to test 52 abstract classes of ant.

Column 3 of Table V shows the average number of IMPL classes created for each abstract class. Since the number of IMPL classes constructed for a single abstract class is either 0 or 1, the average of 0.63 can be read as saying that, on average, 63% of the abstract classes could not be fully tested without the creation of the concrete descendant IMPL class. This shows clearly that the majority of abstract classes in the study cannot be fully tested solely through their concrete descendants. However, since it also shows that 37% of classes can be tested in this way, it underlines the advantages of the hybrid approach over the sole use of existing concrete descendants, or the sole use of synthesized concrete descendants. It should also be noted that although 63% of the abstract classes require the IMPL class during testing only 28% of the methods in the abstract classes are tested in the IMPL class.

The highest value in Column 3 is a value of 1.00 for compiere, meaning that one new class had to be created for each abstract class. It is worth noting that this application has the highest *fanin* value for abstract classes in Table III, suggesting a high level of dependencies on its abstract classes. Indeed, its *fanin* value of 42.7 is a significant outlier: the same figure for other applications ranges from 6.2 to 15.8, with an average of 10.26 and standard deviation of 3.24.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr



Comparing the data in Columns 2 and 3 suggests that the number of concrete descendants used in testing is roughly comparable to the number of generated IMPL classes for many applications. The purpose of Columns 4 and 5 is to provide a basic measure of the relative complexity of these classes in terms of the number of methods they contain. As can be seen from the data in Column 4, for each concrete descendant tested, on average, between 3.3 and 29.7 methods were from its parent abstract class, with an average of 12.11 methods across all applications. In contrast, the relative size of the generated IMPL classes is much smaller, ranging from 2.13 to 12.20 methods, with an average of 4.72 methods per class. The highest value in Column 5 of Table V is for jboss, where there are on average 12.2 methods in each IMPL class. From Column 4 it can be seen that jboss had an exceptionally low level of truly inherited methods in its concrete descendants. On average, a concrete descendant in jboss has just 3.3 truly inherited methods, whereas the same figure for the other applications ranges from 7.6 to 29.9 with an average of 12.7 and standard deviation of 5.6.

In summary, the data in Table V shows that abstract classes cannot in general be tested solely through their concrete descendants or any subset of these. However, the data shows that a significant proportion of the functionality of the abstract classes can be tested in this way, and so an approach that solely uses generated concrete children for testing will miss out on this potential re-use. Thus the approach presented, which uses concrete classes when possible, and generated IMPL classes when necessary, offers considerable improvements over these existing approaches.

5.4. Analysis of test orderings

The creation of stubs during integration testing is considered a major cost factor, as a result several integration testing strategies have been developed to minimize the number of stubs used during OO testing [14,16,18,19]. Although these strategies focus on minimizing the number of stubs required during integration testing of classes, the basic ideas used in these strategies can be applied to integration testing of methods. However, it should be noted that unlike the strategies used for classes, the edges in the dependency graphs representing the interactions between methods are not labelled or weighted.

One of the key aspects of each integration test strategy is the selection criteria used to identify which edge(s) to remove if the dependency graph is not acyclic. For the class-based testing strategies the selection criteria involve some combination of weighted edges based on the edge labels and the in-degree and out-degree of each vertex in the dependency graph. In the experiments presented in this section different combinations of the in-degree and out-degree of each vertex in the dependency graph are used. In Figure 5 the edge selection criteria identifies the edge where the product of the in-degree of the source vertex and out-degree of the target vertex is a maximum.

Recall that generating a test order for the truly inherited methods in the concrete descendant classes of an abstract class requires two main steps, (1) generating a near-minimal set cover for the truly inherited methods, and (2) generating the test order for the methods in the abstract class. The set cover may require the use of a class (IMPL) created solely for the purpose of testing some of the methods in the abstract class not covered in the concrete descendants. While experimenting with different edge selection criteria on the call graph for the example in Figure 1, it was realized that the number of stubs required to test the methods changed, and the methods and their respective stubs were located in different classes. Figure 7 shows a dependency graph representing the test order when the edge selected is the one where the source vertex has a maximum in-degree. This edge selection

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Figure 7. Dependency graph showing one test order as tested in A's concrete subclasses, using the maximum source vertex in-degree edge selection criterion.

criterion requires the use of three stubs: (1) stub `B2.a1(P)` when testing method `A_IMPL.a5()`, (2) stub `B2.a6()` when testing method `B2.a8(int)`, and (3) stub `B2.a1(P)` when testing method `B2.a6()`.

Unlike the test order shown in Figure 5 where both the method and the stub are in the same class, `B2`, it is possible for the method and its required stub to be in different classes as shown in Figure 7, e.g. case (1) where the method is in `A_IMPL` and stub in `B2`. In such cases it may be better to test the method and the required stub in the `IMPL` class in order to reduce the effort of creating stubs in different classes. Moving the methods and required stubs into the `A_IMPL` class would result in a modified test order where `a1(P)`, `a6()`, and `a8(int)` are tested in `A_IMPL`.

Experiments were performed on the application suite in Table I using six different edge selection criteria. The results of applying three of the edge selection criteria are shown in Table VI. These three edge selection criteria used were the maximum source in-degree, the minimum source in-degree and maximum product of the source in-degree and target out-degree. The other three edge selection criteria not shown in Table VI are: minimum out-degree of the source vertex, minimum total of the source vertex in-degree and target vertex out-degree of the source vertex, and maximum total of the source vertex in-degree and target vertex out-degree of the source vertex. The results for the latter three criteria are not presented in the paper as they did not produce a significant variance from the data in Table VI.

The data in Table VI compares the three chosen edge selection criteria for the 12 applications that require stubs during testing; the names of these applications are shown in Column 1. Columns 2 and 3 of Table VI show the data for the edge selection criterion 1 which identifies the edge where the source vertex has a maximum in-degree. Columns 4 and 5 of Table VI show the data for criterion 2 where the edge selected has the source vertex with the minimum in-degree. Columns 6 and 7 of Table VI show the data for criterion 3 where the edge selected is the one with the maximum value for the product of the source vertex in-degree and target vertex out-degree.

In Table VI two metrics are used to measure the effectiveness of each edge selection criteria:

- $|S|$ is a count of the number of stubs required during testing.
- $|(M_k, S_k) \notin C_j|$ is a count of the number of methods and their required stubs not in the same concrete class.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

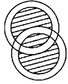


Table VI. Test ordering strategies using different edge selection criteria: (1) maximum source in-degree, (2) minimum source in-degree, and (3) maximum product of the source in-degree and target out-degree.

	Criterion 1		Criterion 2		Criterion 3	
Application	S	$(M_k, S_k) \notin C_j$	S	$(M_k, S_k) \notin C_j$	S	$(M_k, S_k) \notin C_j$
ant	6	4	6	4	6	4
columba	1	0	1	1	1	1
compiere	3	1	2	1	2	1
derby	8	4	8	4	8	4
eclipse-SDK	55	25	49	26	46	25
geronimo	2	1	2	1	2	1
hibernate	2	1	2	1	2	1
jboss	2	1	2	1	2	1
jre	76	49	74	50	67	42
jtopen	15	7	15	7	15	7
netbeans	36	17	35	15	33	15
sandmark	12	4	12	3	12	5
Average	13.63	7.56 (55%)	13.00	7.13 (56%)	12.25	6.69 (55%)

$|S|$—number of stubs, $(M_k, S_k) \notin C_j$—method and stub not in same concrete class.

The values of these metrics in Table VI show relatively little variance between the three criteria. The main difference is that on average Criterion 3 uses fewer stubs during testing, and for each application also uses fewer or the same number of stubs as the other edge selection criteria.

The results in Table VI show that Criterion 3 is the best of the edge selection criteria used in the experiments with respect to minimizing the number of method stubs used to test the methods of an abstract class. Although Criterion 3 is not significantly better that the other edge selection criteria it supports the results obtained by Briand *et al.* for minimizing the number of class stubs when testing class clusters [14]. That is, when breaking the cycles in a dependency graph a good edge selection criterion should consider using the product of the source vertex's in-degree and target vertex's out-degree of the edge to be removed. The approach used by Briand *et al.* generated a test order for class clusters using the ORD [19]. The inter-method call graph used to capture the dependencies between methods is similar to the strategy used by Briand *et al.* since they only used one type of edge in the ORD.

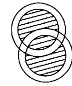
6. CONCLUSION

Testing OO software continues to present many challenges to researchers and industry practitioners. This paper has focused on testing abstract classes that, unlike concrete classes, cannot be instantiated directly, thereby making many of the testing techniques developed to test OO software unusable.

In this paper a strategy for testing abstract classes that makes qualified use of its concrete descendant classes is presented. The strategy involves a lightweight dependency analysis of the inherited methods of the concrete descendants and uses the results of this analysis to generate

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr



an integration test order for the methods in the abstract class. The test order generated seeks to minimize the number of stubs required during testing.

This paper is an extension of earlier work and includes two empirical studies on a corpus of 16 large Java applications and the results have led to the following main conclusions:

1. A high percentage of concrete classes have a direct dependence on abstract classes, even though the number of abstract classes in an application may be relatively small.
2. In general an abstract class cannot be solely tested through its concrete descendants or any such subset.
3. The edge selection criteria for breaking the cycles in an inter-method dependency graph have a small impact on the number of stubs required to test the methods in an abstract class.

6.1. Threats to validity

In this paper two empirical studies are presented to support a testing strategy of the features in an abstract class. The first empirical study analysed abstract classes to motivate the need to adequately test the features of an abstract class and the second study investigated the pragmatics of using the strategy to test the features of an abstract class. There are several factors that may have affected the results of the experiments and they are presented in this subsection.

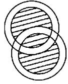
With any empirical study based on selecting software applications, there is a threat to external validity concerning the representativeness of the applications used. The selection of Java applications was based on an existing corpus, which assists with replicating the experiment, but may introduce bias in terms of the application type. In particular, the study is limited to the 16 largest applications based on the number of classes they contain, and it is possible that different results might be achieved for a different selection of applications. In more general terms, it is also possible that results from Java applications, where abstract classes have long been a recognized design idiom, may not be generalizable to other programming languages.

A number of potential threats to construct validity arise from the study of abstract classes presented in Section 4. In the study *fanin* and *scc* metrics are used as measures of dependency in order to test the SAP. While these are common metrics, it should be noted that there are many different ways of measuring dependencies between classes, and it is possible that using different metrics would yield different results. Indeed the work of Briand *et al.* has already noted that many different measures of coupling between classes can be inferred from the basic description [39], and it is thus possible that different metrics could yield different results.

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 the results in this paper. However, the results are partially validated by generating similar numbers in two components of the *AbstractTestJ* tool. For example, the number of abstract classes and the number of concrete direct descendants were generated in both *TaxTOOLJ* and `memberDepend`, which uses the BCEL package. The results generated from both components were consistent. The subsystems of the `abstractTestJ` package (see Figure 6) were validated using the illustrative example that threads the paper and performing a structured walkthrough using the pmd application shown in Table I. The *Dependency Finder* tool [40] was also used to check the number of abstract classes analysed by the *AbstractTestJ* tool, the results were consistent.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr



There are several practical limitations of the study including: (1) finding the class libraries required by the various applications, (2) using the library JGraphT [37], 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 analysed. However, the missing libraries did not affect the number of classes processed by *AbstractTestJ*. 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 attempting to catalog large applications, typically resulting in an out of memory error, and these required some rather *ad hoc* work-arounds.

At a higher level it might be argued that there is a certain cyclic aspect to the approach presented in the paper. Since the approach is based on a code analysis this means that it cannot be performed until the code has been written. Yet, with a Test-Driven Development (TDD) approach, the testing would already have occurred at this stage. However, there are at least three counter arguments to this. First, from a software maintenance perspective it is not unusual to have to deal with a body of code with incomplete test cases. Second, even after a TDD approach has integrated the system, the analysis presented in this paper can act as a final check that this development has proceeded correctly. Finally, the results presented in the previous section underpin the importance of testing either during or after integration, since they highlight the complexities of fully testing code in abstract classes.

6.2. Future work

Future work involves extending the testing strategy for abstract classes and performing empirical studies to identify relationships between the *Stable Abstractions Principle* (SAP) and occurrence of faults.

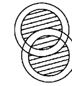
Extending the testing strategy of abstract classes will include developing an optimized approach that not only tests the methods in abstract classes but considers the overall testing effort for all the methods in the concrete descendant classes. In particular, the results of the empirical study to analyse the different test orderings suggest that if a method and its associated stub occur in different concrete descendant classes, it may be more efficient to test both that method and its stub in the concrete descendant class created only for testing purposes. This would mean that all the stubs would be located in a single concrete descendant class.

The authors are currently investigating how the correlation between stability and abstraction impact the occurrence of faults in successive versions of Java applications. The SAP suggests that a good OO design ought to exhibit a high degree of dependencies on abstract classes, and uses this level of dependency as a measure of 'stability'. A longitudinal study of successive versions of Java applications is currently being investigated to see if the stability metric correlates with the actual stability of classes between versions. The study also plans to look into why the implemented methods in an abstract class do not form a complete set cover in the concrete descendants, thereby requiring a synthetic class to be created during testing.

A central theme of this paper and future work is the emphasis on making empirical results replicable and available. While the experimental data is available on request from the authors, there are plans to investigate mechanisms that will allow this data to be made available in a manner that will facilitate other researchers who wish to repeat or extend the experiments in this paper.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr



ACKNOWLEDGEMENTS

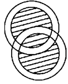
This work was supported in part by the National Science Foundation under grants HRD-0317692 and HRD-0833093.

REFERENCES

1. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns*: *Elements of Reusable Object-Oriented Software*. Addison-Wesley: Reading, MA, 1995.
2. Binder RV. *Testing Object-Oriented Systems*: *Models*, *Patterns*, *and Tools*. Addison-Wesley: Reading, MA, 2000.
3. Weyuker EJ. The evaluation of program-based software test data adequacy criteria. *Communications of the ACM* 1988; **31**(6):668–675.
4. Harrold MJ, McGregor JD, Fitzpatrick KJ. Incremental testing of object-oriented class structures. *Proceedings of the 14th International Conference on Software Engineering*. ACM: New York, 1992; 68–80.
5. Clarke PJ, Babich D, King TM, Power JF. Intra-class testing of abstract class features. *Eighteenth IEEE International Symposium on Software Reliability Engineering*, Trollhattan, Sweden, 2007; 191–200.
6. Melton H, Tempero E. An empirical study of cycles among classes in Java. *Empirical Software Engineering* 2007; **12**(4):389–415.
7. Tempero E, Noble J, Melton H. How do Java programs use inheritance? An empirical study of inheritance in Java software. *European Conference on Object-Oriented Programming*, Paphos, Cyprus, 2008; 667–691.
8. Martin RC. *Agile Software Development*: *Principles*, *Patterns*, *and Practices*. Prentice-Hall: Englewood Cliffs, NJ, 2003.
9. Meyer B. *Object-Oriented Software Construction*. Prentice-Hall PTR: Upper Saddle River, NJ, U.S.A., 1997.
10. Gosling J, Joy B, Steele G, Bracha G. *The Java Language Specification* (3rd edn). Prentice-Hall: Englewood Cliffs, NJ, 2005.
11. McGregor JD, Sykes DA. *A Practical Guide To Testing Object-Oriented Software*. Addison-Wesley: Reading, MA, 2001.
12. Thuy N. Testability and unit tests in large object-oriented software. *Fifth International Software Quality Week*, Software Research Institute, 1992; 1–9.
13. IEEE Std 61012-1990(R2002). *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Computer Society Press: Silver Spring, MD, 1990.
14. Briand LC, Labiche Y, Wang Y. An investigation of graph-based class integration test order strategies. *IEEE Transactions on Software Engineering* 2003; **29**(7):594–607.
15. Hanh VL, Akif K, Traon YL, Jézéquel JM. Selecting an efficient OO integration testing strategy: An experimental comparison of actual strategies. *ECOOP '01*: *Proceedings of the 15th European Conference on Object-Oriented Programming*. Springer: London, U.K., 2001; 381–401.
16. Le Traon Y, Jéron T, Jézéquel JM, Morel P. Efficient object-oriented integration and regression testing. *IEEE Transactions on Reliability* 2000; **49**(1):12–25.
17. Lloyd EL, Malloy BA. A study of test coverage adequacy in the presence of stubs. *Journal of Object Technology* 2005; **4**(5):117–137.
18. Abdurazik A, Offutt J. Using coupling-based weights for the class integration and test order problem. *The Computer Journal* 2009; **52**(5):557–570. DOI: http://dx.doi.org/10.1093/comjnl/bxm054.
19. Kung D, Gao J, Hsia P, Toyoshima Y, Chen C. A test strategy for object-oriented programs. *Nineteenth International Computer Software and Applications Conference*, Los Alamitos, CA, 1995; 239–244.
20. Mackinnon T, Freeman S, Craig P. Endo-testing: Unit testing with mock objects. *Extreme Programming Examined*, Succi G, Marchesi M (eds.). Addison-Wesley Longman Publishing Co.: Boston, MA, 2001; 287–301.
21. Freese T, Tremblay H. Easymock. 2009. Available at: http://easymock.org/ [14 October 2009].
22. Kong L, Yin Z. The extension of the unit testing tool Junit for special testings. *First International Multi-Symposium on Computer and Computational Sciences*, vol. 2, Washington, DC, 2006; 410–415.
23. Tai KC, Daniels FJ. Interclass test order for object-oriented software. *Journal of Object-Oriented Programming* 1999; **12**(4):18–25.
24. Standard Performance Evaluation Corporation. SPEC releases SPEC JVM98, first industry-standard benchmark for measuring Java virtual machine performance. Available at: http://www.specbench.org/osg/jvm98/press.html [19 August 1998].
25. Dujmović JJ. Universal benchmark suites—A quantitative approach to benchmark design. *Performance Evaluation and Benchmarking with Realistic Applications*, Eigenmann R (ed.). MIT Press: Cambridge, MA, 2000; 257–287.
26. Maletic J, Marcus A. CFB: A call for benchmarks—For software visualization. *Second IEEE Workshop of Visualizing Software for Understanding and Analysis*, Amsterdam, The Netherlands, 2003; 108–113.

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr



27. Sim S, Easterbrook S, Holt R. Using benchmarking to advance research: A challenge to software engineering. *International Conference on Software Engineering*, Portland, OR, U.S.A., 2003; 74–83.
28. Martin R. OO design quality metrics: An analysis of dependencies. Available at: http://www.objectmentor.com/resources/articles/oodmetrc.pdf, comp.lang.c++ [23 August 1994].
29. Martin RC. The tipping point: Stability and instability in OO design. Dr Dobb's Portal. Available at: http://www.ddj.com/dept/architect/184415285 [6 November 2009].
30. Melton H, Tempero E. A simple metric for package design quality. *Technical Report UoA-SE-2005-7*, Software Engineering, University of Auckland, September 2005.
31. Babich D, Chiu K, Clarke PJ. TaxTOOLJ: A tool to catalog Java classes. *Eighteenth International Conference on Software Engineering and Knowledge Engineering*, San Francisco, U.S.A., 2006; 375–380.
32. Clarke PJ, Malloy BA. Using a taxonomy to analyze classes during implementation-based testing. *Eighth IASTED International Conference on Software Engineering and Applications*, MIT Cambridge, U.S.A., 2004; 288–293.
33. Clarke PJ, Malloy BA. A taxonomy of OO classes to support the mapping of testing techniques to a class. *Journal of Object Technology* 2005; **4**(5):95–116.
34. Liskov BH, Wing JM. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems* 1994; **16**(6):1811–1841.
35. Cormen TH, Leiserson CE, Rivest RL, Stein C. *Introduction to Algorithms* (2nd edn). MIT, McGraw-Hill: Cambridge, MA, 2001.
36. Tarjan R. Depth-first search and linear graph algorithms. *SIAM Journal on Computing* 1972; **1**(2):146–160.
37. JGraphT Development Team. JGraphT 0.7.0. Available at: http://jgrapht.sourceforge.net/ [15 July 2006].
38. Dahm M, van Zyl J, Haase E, Brosius D, Curdt T. Byte Code Engineering Library v5.2. Available at: http://jakarta.apache.org/bcel/ [23 June 2006].
39. Briand L, Daly J, Wüst J. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering* 1999; **25**(1):91–121.
40. Tesser J. Dependency Finder, 2002. Available at: http://depfind.sourceforge.net [23 June 2006].

Copyright © 2010 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr