Class Mutation: Mutation Testing for Object-Oriented Programs

Sunwoo Kim John A. Clark John A. McDermid

High Integrity Systems Engineering Group
Department of Computer Science
University of York, York
YO10 5DD, United Kingdom
{sunwoo,jac,jam}@cs.york.ac.uk

Abstract. The program mutation is a testing technique that assesses the quality of test input data by examining whether the test data can distinguish a set of alternate programs (representing specific types of faults) from the program under test. We have extended the conventional mutation method to be applicable for object-oriented (OO) programs. The method, termed *Class Mutation*, is a form of OO-directed selective mutation testing that focuses on plausible flaws related to the unique features in OO (Java) programming. This paper introduces the Class Mutation technique and describes the results of the case study performed to investigate the applicability of the technique.

1 Introduction

The mutation method [DeMillo78] is a fault-based testing strategy that measures the quality/adequacy of testing by examining whether the test set (test input data) used in testing can reveal certain types of faults. Unlike other fault-based strategies that directly inject artificial faults into the program, the mutation method generates simple syntactic deviations (mutants) of the original program, representing 'typical' programming errors. For example, a mutation system replaces an arithmetic operator (say +) in the original program with other operators (such as –, *, and /), which is intended to represent the programmer using a wrong operator. If a test set can distinguish a mutant from the original program (i.e. produce different execution results), the mutant is said to be *killed*. Otherwise, the mutant is called a *live* mutant. A mutant may remain live because either it is equivalent to the original program (i.e. it is functionally identical to the original program although syntactically different) or the test set is inadequate to kill the mutant. If the mutant is an *equivalent mutant*, it would always produce the same output, hence it cannot be killed. If a test set is inadequate, it can be improved by adding test cases to kill the (non-equivalent) live mutant. A test set that can kill all non-equivalent mutants is said to be *adequate*.

Our interest is in using the mutation technique to examine the adequacy of test data for object-oriented programs. Obviously, the traditional mutation method can be applied to OO programs. However, using an existing mutation system as it is may not

be sufficient to adequately test OO programs because the existing mutation systems were developed in non-OO programming environments[1]. That is, the common programming errors that the traditional mutation systems model were derived from non-OO programming experience and thus they do not consider some kinds of errors likely to appear in OO programs.

In addition, the differences and new features in OO programming are likely to change the requirements for mutation testing. For instance, the conventional mutation systems make mutants of expressions, variables, and statements but do not mutate type and component (e.g. a data structure) declarations. Agrawal explains "once we regard declarations to be program entities that state facts, we cannot mutate them because we have assumed that there is no scope for any syntactic aberration [Agrawal89]." Traditional programming simply makes use of the built-in types and entities of a language which are unlikely to contain many errors, so you wouldn't get much benefit by changing the declarations of those pre-defined program entities. However, OO programs are composed of user-defined data types (classes) and references to the user-defined types. It is very likely that user-defined components contain many defects such as mutual dependency between members/classes, inconsistencies or conflicts between the components developed by different programmers, etc.

In object-oriented systems, mutation testing should also consider the relationships between components even though it is presently aimed at testing a single method or a class. For example, traditional mutation uses pre-fixed type compatibility (e.g. all arithmetic types are considered compatible) to replace a variable with other variables of compatible types. This is reasonable where there are only pre-defined built-in types. However, type compatibility of OO programs should be flexible because programmers can declare as many types as they want, which will change class structures. Compatibility thus needs to be dynamically determined by considering cluster structure at the time mutation is performed.

The effectiveness of mutation testing, like other fault-based approaches, heavily depends on the types of faults the mutation system is intended to represent, as they actually decide what to test and point out where analysis should be done. In our opinion, it is mainly the flaws related to OO-specific features such as inheritance, polymorphism, and so on that the current mutation systems fail to adequately handle. To address this concern we have developed a method called *Class Mutation* which particularly targets plausible faults that are likely to occur due to OO-unique features in OO programming. For empirical study, we have applied the Class Mutation technique to an IBM product called *Product Starter Kit (PSK) for Java 1.0* which provides classes (helper classes) to facilitate development of production quality Java applets and applications. The test sets for the product were generated according to the traditional control-flow and data-flow testing methods and the adequacy of the test sets was assessed with the Class Mutation technique.

---

[1] As far as we are aware, the mutation systems for Fortran, Cobol, C, and Ada are available. Apart from Ada, they are non-OO languages, and strictly speaking Ada is not an object-oriented but object-based language.

In Section 2, we give a description of the Class Mutation technique. The examples and rationale supporting the technique are illustrated with respect to each OO feature. Section 3 summarises the results of our case study performed to evaluate Class Mutation. Finally, our conclusions and future work are presented in Section 4.

2 Class Mutation

Class Mutation is a mutation technique for OO (Java) programs. The main difference from the traditional mutation method is that it is targeted at plausible faults related to OO-specific features that Java provides – class declarations and references, single inheritance, information hiding, and polymorphism. Faults are introduced into the program by a set of new *mutation operators* (predefined program modification rules). Deriving the plausible flaws related to OO-unique features is presented in other papers [Kim99a][Kim99b]. This paper focuses on illustrating each Class Mutation operator that produces OO-directed faults.

2.1 Polymorphic types

In OO systems, it is common for a variable to have polymorphic types. That is, a variable may at runtime refer to an object of a different type from its declaration and to objects of different types at different times. This raises the possibility that not all objects that become attached to the same variable correctly support the same set of features. This may also cause runtime type errors which cannot always be detected at compile time. Two mutation operators, CRT and ICE, were designed to address this feature.

- **CRT (Compatible Reference Type replacement)**

This operator replaces a reference type with compatible types in a cluster[2]. The compatible types are the names of other types (classes) that meet the widening/narrowing reference conversion rules in the Java language specification [Gosling96]. For instance, the class type `S` can be replaced with the class type `T` provided that `S` is a subclass of `T`, or `S` can be replaced with the interface type `K` provided that `S` implements `K`.

```
class T { … }
class S extends T implements K { … }
class U extends S { … }
interface K { … }
```

The original code:

```
S s = new S();
```

[2] A cluster is a number of classes related to each other in terms of the relationships in the OO paradigm such as inheritance, class nesting, aggregation, association, etc.

CRT Mutants:

```
a) T s = new S();
b) K s = new S();
c) U s = new S();
```

Although we apply only legal conversion rules for the compatible types to be replaced, it may cause compilation errors sometimes. In this example, mutant c) cannot be executed. However, if the original code was `S s = new U();` mutant c) would have no problem.

- **ICE (class Instance Creation Expression changes)**

While CRT handles the declared types, the ICE operator was designed to change the runtime type of an object. Java 'instance creation expression' creates an object of the class specified in the expression. The ICE operator replaces the class name in 'instance creation expression' with compatible class names. This results in calling the constructors of compatible types, which will create the objects of the replaced types. For example, the original code above can have two mutants created by the ICE operator.

ICE Mutants:

```
a) S s  = new T();
b) S s  = new U();
```

Mutant a) will call the constructor in class `T` creating an object of type `T` (resulting in compilation errors), while mutant b) creates an object of type `U` through the constructor of class `U`.

The CRT and ICE operators are intended to distort the program by deliberately giving a wrong but compatible type so that testers can examine differences in compatible types and the possibility of mistakenly using an unintended type. It has been observed that in OO systems the type replacement between compatible types often do not show obvious difference in the behaviour of the system as they are very closely related to each other [Kim99a]. Thus it is likely that test cases fail to distinguish one type from another which indicates more opportunities for type errors. The CRT and ICE operators will force the tester to generate test cases that are strong enough to differentiate a user-defined type from its compatible types. We also expect that the operators help testers discover possible runtime type errors in earlier stages. Meyers states the difficulty in preventing runtime errors and suggests that "whenever you can, push the detection of an error back from runtime to link-time, or, ideally, to compile time [Meyers96]." By replacing a type with other types a runtime type error might be shifted forward to compile time so that testers can catch them during compilation.

The ICE operator can also be used to handle object initialisation. It seems that initialisation faults are the most frequently made errors irrespective of language paradigm. For example, errors such as the failure to set proper initial states (e.g. no value or a wrong value is assigned to an attribute variable) or failure to reinitialise a variable are often found in OO systems. The ICE operator is basically intended to call all possible constructors for an object creation by replacing each class instance

creation expression with other possible expressions. The ICE operator thus may change the initial states of an object as well as the type of the created object. For example, the following code shows two more mutants created by the ICE operator in addition to the ones shown above.

```
c) S s = new S(1, "login");
d) S s = new U(1, "login", "Msg");
```

Mutant c) calls the constructor of the same type as the original code but with different parameter values, so it will change the initial state of the created object. Mutant d) creates an object of a different type with different initial states.

The intent is to create an object with all possible ways provided by programs and examine the difference made by different initial states, which might highlight any errors in object creation/initialisation.

2.2 Method Overloading (parametric polymorphism)

A class type may have more than one method with the same name as long as they have different signatures. There is more possibility of an unintended method being called, even if the correct method name is given, when several versions of the same name method are available. In order to handle the method overloading feature, we manipulate parameters in method declarations and arguments in method invocation expressions. The CRT operator contributes to examining this feature as it changes the types of method parameters in method declarations. We also propose the POC, VMR, AOC, and AND operators for the method overloading feature. The intent of all these operators is distorting the intended method declarations or method invocation expressions, to check if one of the overloaded methods is mistakenly invoked instead of the intended one, as it happens to be better matched.

- **POC (method Parameter Order Change)**

The POC operator changes the order of parameters in method declarations if the method has more than one parameter. For example, the `LogMessage` class in our case study has five overloading constructors, and two of them are:

```
1. public LogMessage(int level, String logKey, Object[]
inserts) {…}
2. public LogMessage(int level, String logKey, Object
insert) {…}
```

The POC operator creates a mutant of the first constructor as below – it swaps the first and second parameters of the constructor.

POC mutant:

```
public LogMessage(String logKey, int level, Object[]
inserts) {…}
```

This mutant program is executed without compilation errors in spite of the fact that the types of the swapped parameters are totally different (i.e., non-compatible). The reason is that the instance creation expressions that call the first constructor in the original code are directed to the second constructor when the mutant is executed, because the second constructor is now better fitted. It is possible because arrays can be assigned to variables of type `Object` in Java. `Object` is the root class in Java, which means that all objects can be assigned to a variable of type `Object`. This example shows that there is a possibility of invoking a wrong constructor/method among the overloaded constructors/methods due to an unintended parameter type conversion.

- **VMR (oVerloading Method declaration Removal)**

This operator removes a whole method declaration of overloading/overloaded methods, in turn. If there is an error of invoking one of the overloaded methods instead of the intended method, the program will work fine even though the intended method is deleted. On the contrary, deleting a method should not cause any problem for the method invocation expressions that are supposed to invoke one of the overloaded methods but not the deleted method. In this way, it can check whether the right method is invoked for the right method invocation expressions. The VMR operator can also provide coverage for the method overloading feature – i.e., checking if all the overloading/overloaded methods are invoked at least once – because test data must reference the method in order to notice that that method has been deleted.

- **AOC (Argument Order Change)**

The AOC operator changes the order of arguments in method invocation expressions, if there is more than one argument. For example, the following mutant produced by the AOC operator represents the error of 'a wrong argument order in a method invocation expression'. As both arguments have the same type (Java `String`), the order change in the mutant did not cause compilation problems, and despite the values passed to the parameters of the called method being different, the test sets used in our case study could not kill the mutant. It turns out that none of the test sets generated according to the control-flow and data-flow testing methods explicitly checks trace output, so they fail to discover the change.

The original code: `Trace.entry("Logger", "addLogCatalogue");`
AOC Mutant: `Trace.entry ("addLogCatalogue", "Logger");`

- **AND (Argument Number Decrease)**

The AND operator reduces the number of arguments one by one, if there are more than one argument in method invocation expressions. The following example is a mutant created by the AND operator.

The original code: `Trace.trace(Trace.Event,this,sccsid);`
AND mutant: `Trace.trace(this,sccsid);`

The mutant, although having two arguments instead of three, is successfully compiled because class `Trace` has four different `trace` methods as in the following. The original code calls the first method while the mutant calls the third method.

```
1. public void trace(int level, Object obj, String
traceText) {…}
2. public void trace(int level,String className,String
traceText) {…}
3. public void trace(Object obj, String traceText) {…}
4. public void trace(String className, String
traceText) {…}
```

Despite the fact that the method called by the original code and the method invoked by the mutant are different, our test sets fail to notice this, which indicates inadequacy of the test sets. That is, if there was a real error that mistakenly calls a wrong method among the overloaded methods, it is unlikely that this test data sets are able to detect the error.

2.3 Method Overriding/Hiding in Inheritance

A class type may contain a method with the same name and the same signatures as the method declared in superclasses or superinterfaces. In this case, the method in a subclass overrides the method of a superclass (method overriding/hiding)[3]. When there is more than one method of the same name, it is important for testers to ensure that a method invocation expression actually invokes the intended method. Since a method invoked is determined according to the runtime type of an object (dynamic binding), the ICE operator is related to this feature. In addition, the OMR operator is designed to check that overriding/overridden methods are invoked appropriately.

- **OMR (Overriding Method Removal)**

The OMR operator removes a declaration of an overriding/hiding method in a subclass so that a reference to the overriding method goes to the overridden/hidden method instead. The overriding method in a subclass should have different functionality to the overridden method in a superclass. Otherwise, the programmer would simply inherit the method in the superclass instead of re-implementing it. So, if a test set fails to see any difference whether the overriding method is called or the overridden method is called, it implies that either the current test set is inadequate to handle the method overriding feature or there may be some errors (coincidental correctness). The OMR operator also checks coverage for the method overriding feature – i.e., overriding and overridden methods are invoked by test data at least once.

---

[3] If the methods declared in both a superclass and a subclass are '`static`' it is called 'method hiding,' otherwise it is called 'method overriding'. It is not possible to override an instance method with a static method or hide a static method with an instance method – this would result in compilation errors [Gosling96].

2.4 Field Variable Hiding in Inheritance

A Java class may have two or more fields with the same simple name if they are declared in different interfaces and/or in classes. In this case, the field variables defined in a class hide the fields of the same name declared in superclasses or superinterfaces, and the two fields (hiding and hidden fields) need not have the same type. While this feature is powerful and convenient, it might cause an unintended field being accessed, especially in a long and complex class hierarchy. If the hiding and hidden fields happen to have the same states at certain execution periods, test cases might fail to detect any problem even if an unintended one was referenced. Thus it is important to make sure which field is actually accessed when there is more than one field that can be accessed by the same simple name. The intent of the HFR and HFA operators is to check that hiding and hidden fields are accessed appropriately.

- **HFR (Hiding Field variable Removal)**

The HFR operator removes a declaration of a hiding field variable so the references to that field actually access the field in a superclass or a superinterface. This operator ensures that a test set is able to distinguish referencing a hidden field from referencing a hiding field, which should be one of the test adequacy criteria for the 'field variable hiding' feature. It does not make sense that both hiding and hidden fields are supposed to hold the same states all the time. In that case, programmers would have simply inherited the field. So, if a test set produces the same output even if a hiding field is removed, it indicates the test set is inadequate to give a proper testing for this feature and testers should generate additional test cases.

- **HFA (Hiding Field variable Addition)**

This operator changes an inherited field to a hiding field. That is, it adds field variables that appear in superclasses/superinterfaces into the class under mutation so that the added fields hide those in superclasses/superinterfaces. Both the HFR and HFA operators check the coverage of field variables in the presence of inheritance because test data must accesses the hiding/hidden and inherited fields at least once, in order to notice that one is added or removed. The difference is that the HFA operator checks that inherited fields are accessed at least once whereas the HFR operator checks that hiding/hidden fields are accessed.

In addition, the HFR and HFA operators can help testers uncover any errors in 'field access expression' because removing or adding a hiding field will influence field access expressions that reference the field.

2.5 Information Hiding (Access Control)

In OO programming, a class often contains a number of variables that are interdependent and must be in a consistent state, so it is important to limit access to the attributes and methods that manipulate the attributes. OO languages provide an access control mechanism that restricts the accessibility/visibility of attribute variables and methods. It is an important testing role to make sure that a certain access mode provides and restricts its intended accessibility/visibility at all times

because it could be unexpectedly circumvented in some cases. Meyers provides a C++ example to show how access control can be broken and the restricted members are freely accessible [Meyers96]. Flaws might be caused due to inconsistency in accessibility/visibility (e.g. a field variable has more than one access path and some paths are accessible while others are prohibited). The intended access control can also be broken in connection with other OO features such as inheritance. Several researchers pointed out that the introduction of inheritance can damage information hiding by exposing implementation details to inheriting clients and it might be necessary not only to add new methods, but also to restrict methods that are inherited, when a class is extended.

Java provides four possible access modes – `public, private, protected,` and default (no access modifier specified). We propose the AMC operator that manipulates Java access specifiers to address the information hiding feature.

- **AMC (Access Modifier Changes)**

This operator replaces a certain Java access mode with three other alternatives. For example, a field declaration with a `protected` access mode will have three mutants created by the AMC operator.

The original code:

```
protected Address address;
```

AMC Mutants:

```
public Address address;
Private Address address;
Address address; //default (no access modifier defined)
```

The role of the AMC operator is to guide testers to generate enough test cases for testing accessibility/visibility. That is, in order to make sure that the field `address` is really `protected`, the test set should be able to distinguish it from when its access modifier is `public`/`private`/default mode.

It was observed from the case study that the AMC operator is related to the method overloading feature. In describing the POC operator in Section 2.2, we showed that changes in method parameters cause one of overloaded constructor of class `LogMessage` other than the intended one being invoked. The similar symptom appears due to access modifier changes. The code below shows the AMC operator changes the access modifier of the first constructor of class `LogMessage`, from `public` to `private`.

The original code:

```
1. public LogMessage(int level, String logKey, Object[]
inserts) {…}
2. public LogMessage(int level, String logKey, Object
insert) {…}
```

AMC mutant:

```
1. private LogMessage(int level, String logKey,
Object[] inserts) {…}
```

As the private constructor is not accessible, the first constructor becomes non-available. This causes all the instance creation expressions that initially call the first constructor now use the second constructor – i.e., the second constructor is the best matched one among the 'available' constructors.

2.6 Static/Dynamic States of Objects

Java has two kinds of variables – class and instance variables, which have important differences. The Java runtime system creates one copy of each instance variable whenever an instance of a class is created (dynamic) while class variables are allocated once per class, the first time it encounters the class (static). Each copy of an instance variable is associated with an instance but all instances share the same copy of the class's class variables regardless of the number of instances created of that class. Kaner's error taxonomy [Kaner93] reports that confusion is easy when both types of variables exist. We propose the SMC operator to examine possible flaws in static/dynamic states.

- **SMC (Static Modifier Changes in field declaration)**

The SMC operator removes the '`static`' modifier to change a class variable to an instance variable or adds the modifier to change an instance variable to a class variable. For example, the following code shows two mutants created by the SMC operator.

The original code:

```
1. public static int VALUE = 100;
2. private String s;
```

SMC Mutants:

```
1. public int VALUE = 100;  //static is removed.
2. private static String s; //static is added.
```

The SMC operator is likely to generate many mutants that simply end up generating compilation errors. Nevertheless, the mutants that manage to produce the same output as the original program can draw a tester's attention to locations where subtle errors in object states might occur. For example, our case study shows that several SMC mutants remain live (i.e., the test sets could not make out the difference whether the variable is `static` or not), which indicates that the test sets may fail to discover a wrong use of `static` modifier.

Java also has two types of methods – class (or static) and instance methods. Changing the `static` specifier of a method might give useful information for testing, as it decides whether the method is statically bound (invoke the method of the declared

type of an object) or dynamically bound (invoke the method of the dynamic type of an object).

2.7 Exception Handling

Although exception handling is not a unique feature in the OO paradigm, the availability of the exception mechanism is one of the main differences between OO languages and the traditional languages such as Fortran and Cobol. The most obvious mistake in exception handling is not specifying appropriate exception handlers in the required place, which might lead to an abrupt system halt. Even if they are defined, they may fail to perform the intended exception handling at runtime. Meyers describes several possible flaws in exception handling of C++ programs [Meyers96].

In Java, you either handle an exception (i.e., catch the exception by declaring a `try-catch` block) or propagate it (i.e., declare it to throw in a `throws` statement of a method declaration). The CRT operator introduced in Section 2.1 is related to the exception handling feature, because it replaces the type of an exception class with compatible exception class types. In addition to the CRT operator, we propose the EHR and EHC operators for the feature of exception handling.

- **EHR (Exception Handler Removal)**

The EHR operator modifies the declared exception handling statement (`try-catch-finally`) in two different ways.

1) It removes exception handlers (`catch` clause) one by one when there is more than one handler (it does not matter whether or not the `finally` clause exists).
2) It removes the exception handler and `finally` clause in turn when there exist one handler and the `finally` clause.

The EHR operator is not applied when there is only one handler without `finally` clause because it simply causes compilation errors.

By removing a declared handler the EHR operator intentionally postpones catching an exception and passes it to the nearest handler. The purpose of removing the `finally` block is to make sure that there are test cases that check whether the `finally` clause performs its intended work before an uncaught exception is propagated or a program abruptly stops. This operator also gives coverage of `catch` and `finally` clauses.

- **EHC (Exception Handling Change)**

The EHC operator swaps the way of handling an exception. It changes an exception handling statement to an exception propagation statement, and an exception propagation statement to an exception handling statement. That is, the EHC operator catches the exception that is supposed to be propagated by changing a `throws` declaration to a `try-catch` statement or it propagates the exception that is supposed be caught within the method by changing a `try-catch` statement to a `throws` declaration.

Similarly to the EHR operator, the intent of the EHC operator is to distort the declared exception handling in several ways. For example, the `formatMsg` method in class `LogServiceProvider` formats a message by looking up the corresponding message template in message catalogues (Java Properties files). If no message template is found, an exception is raised, which is caught by the exception handler `MissingResourceException`. The EHC operator removes the exception handler and declares it in the method declaration, so an exception raised will be propagated to a nearest handler instead of being caught in the method.

The original code:

```
String formatMsg(LogMessage msg) { …
    try { …
    } catch (MissingResourceException mre) { ...
    }
…
}
```

EHC mutant:

```
String formatMsg(LogMessage msg)
                 throws MissingResourceException {
…   //try-catch block removed
}
```

In our case study, we generated the test sets according to control-flow and data-flow testing methods. It appears that this mutant is killed by the control-flow test set, but the data-flow test set fails to kill the mutant. The reason is that all the test cases in the data-flow test set call the `formatMsg` method with message keys appearing in the message catalogues all the time. That is, an exception is never raised by the data-flow test set, so it is not affected by the missing exception handler. This example shows that the EHC operator can be used to ensure exception coverage.

Table 1. Class Mutation Operators

Class Mutation operators	Descriptions
CRT (Compatible Reference Type replacement)	Replace a class type with compatible types.
ICE (Instance Creation Expression changes)	Change an instance creation expression with other instance creation expressions of the same and/or compatible class types.
POC (Parameter Order Change)	Change method parameter order in method declarations
VMR (oVerloading Method Removal)	Remove the declaration of an overloading method.
AOC (Argument Order Change)	Change method argument order in method invocation expressions
AND (Argument Number Decrease)	Decrease arguments one by one.
HFR (Hiding Field variable	Remove a field variable declaration when it

Removal)	hides the variable in superclasses.
HFA (Hiding Field variable Addition)	Add a field variable of the same name as the inherited field variable.
OMR (Overriding Method Removal)	The declaration of an overriding method is removed.
AMC (Access Modifier Change)	Replace an access modifier with other modifiers.
SMC (Static Modifier Change)	Add or remove a `static` modifier.
EHR (Exception Handler Removal)	Remove exception handlers one by one.
EHC (Exception Handling Change)	Change an exception handling statement to an exception propagation statement, and vice versa.

3 Evaluation of the Class Mutation operators

Mutation operators are intended to provide coverage as well as to induce errors. Several operators in Class Mutation can be used to provide coverage criteria with regard to OO features. The HFR and HFA operator check coverage for field attributes of a class. The traditional mutation systems have an operator that deletes every statement in a program in turn to achieve statement coverage (every statement in a program is used). Clearly, the statement deletion operator will subsume Class Mutation's HFR operator, as it deletes every statement in code including hiding field variable declarations. The traditional statement deletion operator, however, may not be strong enough to achieve statement coverage in the presence of inheritance because it cannot make sure that an inherited field is accessed at least once. In a subclass, the inherited field does not explicitly appear in the program as code, so there is no statement that the statement deletion operator can delete. The HFA operator in Class Mutation can help ensuring the coverage of the inherited fields by adding a field in a superclass (but not redefined in a subclass) into a subclass. Test data won't be able to kill the HFA mutants unless it access the added field (the originally inherited field). The OMR operator provides coverage checking for overriding/overridden methods in inheritance while the VMR operator checks coverage for overloaded methods.

The difference between these Class Mutation operators and the coverage operators of the traditional mutation systems is that the CM operators are selective (focused on a certain feature) and thus less expensive. For example, the OMR operator does not delete every method declaration in a class. It deletes the declarations of overriding methods only. In addition, the Class Mutation operators consider the relationship between the classes in checking coverage while the traditional operators take account of a single unit only.

The mutants created by the AOC, AND, and POC operators tend to get high killed rates (the majority of the mutants created by these operators is killed), yet the remaining live mutants often provide useful information for analysing flaws related to the overloading feature. The examples in Section 2.2 show that mistakes in method

declarations and/or method references often remain hidden, because one of the overloaded methods is used instead of the intended one rather than explicitly causing compilation errors. Thus there are more possibilities of coincidental correctness, which gives a test adequacy tip – test data, to be adequate in the presence of method overloading feature, should include test cases that recognise the difference in calling a method from calling all the other overloaded methods. The POC operator might be related to the method overriding feature (inheritance) as well as the method overloading feature, because the change in method parameters may cause that an overloading method becomes an overriding method, or an overriding method becomes an overloading method.

The AMC operator forces a test data set to include some package-level test cases, which would improve the quality of the test data and probably be necessary if the class package is to be released outside. The example in Section 2.5 shows that the AMC operator can be related to the method overloading feature as well as the originally intended information hiding feature. The AMC operator also helps checking coverage of field variables. If a variable is never referenced, it will remain unaffected whatever modifier it is given. The main shortcoming of the AMC operator is that applying the AMC operator is expensive and laborious (particularly analysing the live mutants and generating test data to kill them) because it is likely to produce a large number of mutants. How much access restrictions or security checks is required for a program or its data varies with the application. For many systems, or for some parts of a system, it may not be very important whether a program entity is `public` or `protected`. We therefore suggest that testers should use this operator selectively for where access control is critical.

4 Conclusion

Developing a means of assessing how good generated test sets are is an important testing subject. Testers cannot be certain whether they have performed 'adequate' testing for their systems where there are no criteria or standards for measurement available. The mutation method is a promising measurement method for test data adequacy, but its effectiveness for OO programs is challenged because the current mutation systems fail to reflect the OO-unique features. In this paper we have extended the traditional mutation method by proposing a set of mutation operators that are intended to represent plausible flaws related to the unique features in OO (Java) programs. The Class Mutation technique can be used in itself as a form of OO-directed selective mutation testing or it can be integrated with the conventional mutation systems. Most of the popular OO languages such as C++ and Java are actually hybrid languages that combine the source of errors, inherent to both non-OO and OO programming styles. Thus an integrated mutation system should be used to give an adequate mutation testing for the programs written with those languages. The development of an integrated mutation method for Java programs is currently under way.

Acknowledgement
The authors would like to thank Adrian Colyer at IBM Hursley UK for providing the product for the case study.

5 References

[Agrawal89] Agrawal H. et al., Design of Mutant Operators for the C Programming Language, SERC-TR-41-P, Software Engineering Research Center, Purdue University, 1989.
[Delamaro97] Delamaro M. et al., Integration Testing Using Interface Mutations, 1997.
[DeMillo78] DeMillo R. A., Lipton R. J., and Sayward F. G., "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer*, v.11, no. 4, pp. 34-41, April 1978.
[Gosling96] Gosling J. et al., The Java Language Specification, Addison-Wesley, 1996.
[Kaner98] Kaner C. et al., Testing Computer Software, 2nd ed., Int. Thomson Publishing Company, 1998.
[Kim99a] Kim S., Clark J., McDermid J., Assessing Test Set Adequacy for Object-Oriented Programs Using Class Mutation, 28 JAIIO: Symposium on Software Technology (SoST`99), Sept. 1999.
[Kim99b] Kim S., Clark J., McDermid J., The Rigorous Generation of Java Mutation Operators Using HAZOP, to be published in 12th International Conference Software & Systems Engineering and their Applications (ICSSEA`99), Dec. 1999.
[King91] King K.N., Offutt A., A Fortran Language System for Mutation-Based Software Testing, Software Practice and Experience, 21(7): 686-718, July 1991.
[Marick95] Marick B., The Craft of Software Testing, Prentice-Hall, 1995.
[Meyers92] Meyers S., Effective C++: 50 Specific Ways to Improve Your Programs and Designs, Addison-Wesley, 1992.
[Meyers96] Meyers S., More Effective C++: 35 New Ways to Improve Your programs and Designs, Addison-Wesley, 1996.