Inter-Class Mutation Operators for Java

Yu-Seung Ma
Division of Computer Science
Dept of EE and CS
Korea Adv Inst of Sci & Tech
ysma@salmosa.kaist.ac.kr

Yong-Rae Kwon
Division of Computer Science
Dept of EE and CS
Korea Adv Inst of Sci & Tech
kwon@salmosa.kaist.ac.kr

Jeff Offutt*
Dept of Info and Soft Engr
SE Research Lab
George Mason University
ofut@ise.gmu.edu

Abstract

The effectiveness of mutation testing depends heavily on the types of faults that the mutation operators are designed to represent. Therefore, the quality of the mutation operators is key to mutation testing. Mutation testing has traditionally been applied to procedural-based languages, and mutation operators have been developed to support most of their language features.

Object-oriented programming languages contain new language features, most notably inheritance, polymorphism, and dynamic binding. Not surprisingly, these language features allow new kinds of faults, some of which are not modeled by traditional mutation operators. Although mutation operators for OO languages have previously been suggested, our work in OO faults indicate that the previous operators are insufficient to test these OO language features, particularly at the class testing level. This paper introduces a new set of class mutation operators for the OO language Java. These operators are based on specific OO faults and can be used to detect faults involving inheritance, polymorphism, and dynamic binding, thus are useful for inter-class testing. An initial Java mutation tool has recently been completed, and a more powerful version is currently under construction.

1. Introduction

Mutation testing, originally proposed in 1978 [17, 11], is a fault-based testing technique that measures the effectiveness of test cases. Mutation testing is based on the assumption that a program will be well tested if all simple faults are detected and removed. Simple faults are introduced into the program by creating a set of faulty versions, called *mutants*. Test cases are used to execute these mutants with the goal of causing each mutant to produce incorrect output. A test case that distinguishes the program from its mutant is considered to be *effective* at finding faults in the program. The effectiveness of mutation testing, like other fault-based approaches, depends heavily on the types of faults that the mutation system is designed to represent. Since mutation testing uses mutation operators to implement faults, the quality of the mutation operators is crucial to the effectiveness of mutation testing.

Although mutation testing has a rich history, most mutation operators have been developed for procedural programs. Object-oriented (OO) programs tend to be structured rather differently from conventional programs. Method bodies of a well designed OO component are generally shorter than a conventional implementation and interactions between methods of the class or other classes are increased. Moreover, OO languages contain new features such as encapsulation, inheritance, and polymorphism. In previous research [23], we have found that these features introduce the potential for new faults. Therefore, existing mutation operators for procedural programming languages are not sufficient for programs written in OO languages and new OO-specific language operators are needed.

Some research has already been done on the development of mutation operators for OO software. Kim et al. developed a set of class mutation operators [15, 16], resulting in 13 class mutation operators for Java programs. These operators were developed from the OO-specific language features and were intended to be "selective" [19]. Chevalley [4] has developed three additional class mutation operators based on his programming experience. The faults modeled by these operators are not general; they can be application-specific or programmer-specific. Therefore, to execute mutation testing with these operators, they should be selected based on the characteristic of the program to be tested. These previous attempts suffered from not having a general fault model available. These previous OO mutation operators do not handle several fault types and did not handle all Java language features. We also assert that initial mu-

*This work is supported in part by the U.S. National Science Foundation under grant CCR-98-04111.

13th International Symposium on Software Reliability Engineering, Annapolis, MD, November 2002.

tation operators should not be selective, rather they should be comprehensive. Selective mutation operators should be chosen based on empirical evidence of the effectiveness operators.

This paper presents a comprehensive set of class mutation operators for Java that corrects the problems with previous Java operators. This includes previously developed mutation operators, modifications to previously developed mutation operators, and a number of new operators. Section 2 of this paper discusses general issues involved in using mutation testing for object-oriented languages. Sections 4 and 5 describes the class mutation operators for Java and relates this set of operators to previous sets. Finally, Section 6 gives a summary and discusses plans for applying these operators.

2. Using Mutation Testing to Test OO Software

A class is usually regarded as the basic unit of OO testing. Harrold and Rothermel [12] define three levels of testing: (1) *intra-method testing*, in which tests are constructed for individual methods; (2) *inter-method testing*, in which multiple methods within a class are tested in concert; and (3) *intra-class testing*, in which tests are constructed for a single class, usually as sequences of calls to methods within the class. Gallagher and Offutt [10] added *inter-class testing*, in which more than one class is tested at the same time. Following this terminology, we classify faults in classes as occurring at the intra-method level, the inter-method level, the intra-class level, and the inter-class level. Below, we briefly describe each level in terms of faults and describe some considerations for mutation testing.

- **Intra-method Level Faults**
  Intra-method level faults occur when the functionality of a method is implemented incorrectly. A method in a class corresponds to the unit of the conventional program testing. We assume that traditional mutation operators for procedural programs [1, 8, 22] will suffice for this level.

- **Inter-method Level, Intra-class Level Faults**
  Inter-method and intra-class level faults are made at the interactions between pairs of methods of a single class or between pairs of methods that are not part of a class construct in non-OO languages. Because methods are getting smaller and interactions among methods are increasingly encoding the design complexity [27], integration testing is becoming more important. Interface mutation [8] has been proposed to evaluate how well the interactions between various units have been tested. Interface mutation is an extension of mutation testing and is applicable to software systems composed of interacting units. The work in this paper does not address this level of testing.

- **Inter-class Level Faults**
  Inter-class level faults include faults that occur due to the object-oriented specific features such as encapsulation, inheritance, polymorphism, and dynamic binding. Kim et al. [15, 16] and Chevalley [4] have introduced class mutation operators to help detect faults at this level. This paper uses a comprehensive list of object-oriented faults to define inter-class level mutation operators, using and extending the previous operators.

In this paper, we develop class mutation operators for inter-class level faults in the language Java. They specifically target features of the language that are used to integrate classes, including method calls, access, overloading, inheritance, and polymorphism. Because the number of language features are fewer, there are significantly fewer mutants for inter-class faults than for intra-method faults, which most mutation tools have targeted. Most of these mutation operators will be the same in other OO languages, although there are some features that are peculiar to Java. Before introducing the class level mutation operators for Java, we examine the Java OO language features.

2.1. Java Object-oriented Specific Language Features

Encapsulation is an abstraction mechanism to implement information hiding. Information hiding is a design technique that attempts to protect certain aspects of the design from specific parts of the implementation. Encapsulation allows objects to restrict access to their member variables and methods by other objects. Java supports four distinct access levels for member variables and methods: private, protected, public, and default (also called package). These access levels are poorly understood by many programmers, and often not carefully considered during design, so they are a rich source of faults.

Java supports four distinct access levels for member variables and methods: `private`, `default` (also called package), `protected`, and `public`. Table 1 summarizes these access levels. A *private* member is available only to the class in which it is defined. If access is not specified, the access level defaults to *package*, which allows access to classes in the same package, but **not** subclasses in other classes. A *protected* member is available to the class itself, subclasses, and classes in the same package. A *public* member is available to any class in any inheritance hierarchy or package (the world).

Java does not support multiple class inheritance, thus every class has only one immediate parent. A subclass inherits

Table 1. Java's Access Levels

Specifier	Same Class	Same package subclass	Same package non-subclass	Different package subclass	Different package non-subclass
private	Y	n	n	n	n
package	Y	Y	Y	n	n
protected	Y	Y	Y	Y	n
public	Y	Y	Y	Y	Y

variables and methods from its parent and all of its ancestors. The subclass can use these members as defined, or it can override the methods or hide the member variables. Subclasses can add new variables and methods. Subclasses can also explicitly use their parent's variables and methods using the keyword "super" (`super.methodname();`). Java's inheritance allows method overriding, variable hiding, and class constructors.

Method overriding allows a method in a subclass to have the same name, arguments and result type as a method in its parent. Overriding allows subclasses to redefine inherited methods. The child class method has the same signature, but a different implementation.

Variable hiding is achieved by defining a variable in a child class that has the same name and type of an inherited variable. This has the effect of hiding the inherited variable from the child class. This is a powerful feature, but it is also a potential source of errors.

Class constructors are not inherited in the same way other methods are. To use a constructor of the parent, we must explicitly call it using the *super* keyword. The call must be the first statement in the derived class constructor and the parameter list must match the parameters in the argument list of the parent constructor.

One important thing to note is that if *super* does not appear as the first statement of a constructor body, the Java compiler inserts an implicit call *super()* to the constructor. Thus, the **default** constructor (with no parameters) is invoked unless we specifically invoke a parameterized constructor for the parent.

Polymorphism refers to the ability of two or more objects belonging to different classes to respond to the same message in different class-specific ways. Java supports two versions of polymorphism, attributes and methods, both of which use dynamic binding. Each object has a *declared type* (the type in the declaration statement, that is, "*Parent P;*") and an *actual type* (the type in the instantiation statement, that is, "*P = new Child();*", or the assignment statement, "*P = Pold;*"). The actual type can be from any type that is a descendant of the declared type.

A *polymorphic attribute* is an object reference that can take on various types. At any location in the program, the type of the object reference can be different in different executions. A *polymorphic method* can accept parameters of different types by having a parameter that is declared of type *Object*. Polymorphic methods are used to implement *type abstraction* (templates in C++ and generics in Ada).

Overloading is the use of the same name for different constructors or methods in the same class. They must have different *signatures*, or list of arguments. Overloading is easily confused with overriding because the two mechanisms have similar names and semantics. Overloading occurs with two methods in the same class, whereas overriding occurs between a class and one of its descendants.

In Java, member variables and methods can be associated with the class rather than with individual objects. Members associated with a class are called *class* or *static variables* and *methods*. The Java runtime system creates a single copy of a static variable the first time it encounters the class in which the variable is defined. All instances of that class share the same copy of the static variable. Static methods can operate only on static variables; they cannot access instance variables defined in the class. Unfortunately there is some variation in the terminology. In this paper, *instance variables* are declared at the class level and are available to objects, *class variables* are declared with `static`, and *local variables* are declared within methods.

3. OO Specific Faults in Java

The mutation operators in this paper are designed to model possible faults that arise from misuse of OO-specific features. A number of previous papers [9, 2, 23, 16, 15, 4] have investigated OO-specific faults.

The fault model by Offutt et al. [23] defines faults in terms of the language syntax, and therefore is used for defining mutation operators. However, their fault model was restricted to subtype inheritance, thus we incorporate other faults used for mutation operators by Kim et al. and Chevalley [4, 15, 16]. The faults that we consider are briefly presented below; more details are given in the papers.

- Offutt et al.'s faults [23]
  - state visibility anomaly
  - state definition inconsistency (due to state variable hiding)

  - state definition anomaly (due to overriding)
  - indirect inconsistent state definition
  - anomalous construction behavior
  - incomplete construction
  - inconsistent type use
- Kim's faults [15, 16]
  - overloading methods misuse
  - access modifier misuse
  - *static* modifier misuse
- Chevalley and Firesmith [4, 9]
  - incorrect overloading methods implementation
  - *super* keyword misuse
  - *this* keyword misuse
  - faults from programming experience

4. Mutation Operators for Java

We classify the class mutation operators into six groups, based on the language feature that is affected. The first four groups are based on language features that are common to all OO languages. The fifth group includes language features that are Java-specific, and the last group of mutation operators are based on common OO programming mistakes.

1. Information Hiding (Access Control)
2. Inheritance
3. Polymorphism
4. Overloading
5. Java-Specific Features
6. Common Programming Mistakes

Our strategy was to first analyze the list of potential OO faults, then design mutation operators to test for those faults. Several operators were taken from previous research into OO mutation analysis, and the relationships between our operators and the previous operators are detailed in the next section. In the following subsections, we first describe each operator informally, then give an example mutant that can be created from the operator.

4.1. Information Hiding (Access Control)

In our experience in teaching OO software development and consulting with companies that rely on OO software, we have observed that access control is one of the most common sources of mistakes among OO programmers. The semantics of the various access levels are often poorly understood, and access for variables and methods is often not considered during design. This can lead to careless decisions being made during implementation. It is important to note that poor access definitions do not always cause faults initially, but can lead to faulty behavior when the class is integrated with other classes, modified, or inherited from.

Java provides four access levels: public, private, protected, and, if left unspecified, package. The mutation operator AMC is used to test for access control faults.

• **AMC – Access modifier change:** The AMC operator changes the access level for instance variables and methods to other access levels. The purpose of the AMC operator is to guide testers to generate test cases that ensure that accessibility is correct. This mutant can only be killed if the new access level denies access to another class or allows access that causes a name conflict. Because the ability to kill this mutant depends on other characteristics of the software, many of these mutants can be expected to be equivalent.

Original Code		**AMC Mutants**
public Stack s;	Δ	private Stack s;
	Δ	protected Stack s;
	Δ	Stack s;

4.2. Inheritance

Although a powerful and useful abstraction mechanism, incorrect use of inheritance can lead to a number of faults. We define five mutation operators to try to test the various aspects of using inheritance, covering variable hiding, method overriding, the use of *super*, and definition of *constructor*s.

Overriding can cause instance variables that are defined in a subclass to hide member variables of the parent. This powerful feature can cause an incorrect variable to be accessed. Thus it is necessary to ensure that the correct variable is accessed when variable overriding is used, which is the intent of the IHD and IHI mutation operators.

• **IHD – Hiding variable deletion:** The IHD operator deletes a declaration of an overriding, or hiding variable. This causes references to that variable to access the variable defined in the parent (or ancestor). This mimics a common mistake that programmers make. This mutant can only be killed by a test case that is able to show that the reference to the parent variable is incorrect.

Original Code		**IHD Mutant**
class List {		class List {
int size;		int size;
... ...		... ...
}		}
class Stack extends List {		class Stack extends List {
int size;	Δ	// int size;
... ...		... ...
}		}

• **IHI – Hiding variable insertion:** The IHI operator inserts hiding member variables to hide the parent's version of a variable. This mutant can only be killed by a test case that is able to show that the reference to the overriding variable is incorrect.

Original Code

```
class List {
    int size;
    ... ...
}
class Stack extends List {
    ... ...
}
```

Δ

IHI Mutant

```
class List {
    int size;
    ... ...
}
class Stack extends List {
    int size;
    ... ...
}
```

• **IOD – Overriding method deletion:** The ability of a subclass to override a method declared by an ancestor allows a class to modify the behavior of the parent class. When there is more than one method of the same name, it is important for testers to ensure that a method invocation actually invokes the intended method.

The IOD operator deletes an entire declaration of an overriding method in a subclass so that references to the method uses the parent's version. This mutant is killed by a test case that is able to show that the behavior of the parent's method is incorrect.

Original Code

```
class Stack extends List {
    ... ...
    Push (int a) { ... }
}
```

Δ

IOD Mutant

```
class Stack extends List {
    ... ...
    // Push (int a) { ... }
}
```

• **IOP – Overridden method calling position change:** Sometimes, an overriding method in a child class needs to call the method it overrides in the parent class. This may happen if the parent's method uses a private variable *v*, which means the method in the child class may not modify *v* directly. However, an easy mistake to make is to call the parent's version at the wrong time, which can cause incorrect state behavior. The IOP operator moves calls to overridden methods to the first and last statements of the method and up and down one statement.

Original Code

```
class List {
    ... ...
    void SetEnv()
        {size = 5; ... }
}

class Stack extends List {
    ... ...
    void SetEnv() {
        super.SetEnv();
        size = 10;
    }
}
```

Δ

Δ

IOP Mutant

```
class List {
    ... ...
    void SetEnv()
        {size = 5; ... }
}

class Stack extends List {
    ... ...
    void SetEnv() {
        size = 10;
        super.SetEnv();
    }
}
```

• **IOR – Overridden method rename:** The IOR operator is designed to check if an overriding method adversely affects other methods. Consider a method *m()* that calls another method *f()*, both in a class *List*. Further, assume that *m()* is inherited without change in a child class *Stack*, but *f()* is overridden in *Stack*. When *m()* is called on an object of type *Stack*, it calls *Stack*'s version of *f()* instead of *List*'s version. In this case, *Stack*'s version of *f()* may have an interaction with the parent's version that has unintended consequences. The IOR operator renames the parent's versions of these methods so that the overriding cannot affect the parent's method. These mutants can only be killed by a test case that causes different behavior when the overriding (child's) version is not called.

Original Code

```
class List {
    ... ...
    void m() {... f(); ... }
    void f() { ... }
}

class Stack extends List {
    ... ...
    void f() { ... }
}
```

Δ

Δ

IOR Mutant

```
class List {
    ... ...
    void m() {... f'(); ... }
    void f'() { ... }
}

class Stack extends List {
    ... ...
    void f() { ... }
}
```

• **ISK – *super* keyword deletion:** If a subclass hides an instance variable of one of its ancestors, it can still reference the hidden variable by using the *super* keyword. The subclass can also use *super* to invoke a parent's version of a method that has been overridden.

The ISK operator deletes occurrences of the *super* keyword so that a reference to the variable or the method goes to the overriding instance variable or method. The ISK operator is designed to ensure that hiding/hidden variables and overriding/overridden methods are used appropriately.

Original Code

```
class Stack extends List {
    ... ...
    int MyPop( ) {
        ... ...
        return val*super.num;
    }
}
```

Δ

ISK Mutant

```
class Stack extends List {
    ... ...
    int MyPop( ) {
        ... ...
        return val*num;
    }
}
```

• **IPC – Explicit call of a parent's constructor deletion:** Although constructors are not inherited the way other methods are, a constructor of the superclass is invoked when subclasses are instantiated. When we create new objects of a derived class, the default constructor (no arguments) for the parent class is automatically called first, then the constructor of the derived class is called. However, the subclass can

use the *super* keyword to call a specific parent class constructor. This is usually done to pass arguments to one of the parent class's non-default constructors.

The IPC operator deletes *super* constructor calls, causing the default constructor of the parent class to be called. To kill mutants of this type, it is necessary to find a test case for which the parent's default constructor creates an initial state that is incorrect.

Original Code		**IPC Mutant**
class Stack extends List {		class Stack extends List {
... ...		... ...
Stack (int a) {		Stack (int a) {
super (a);	Δ	// super (a);
... ...		... ...
}		}
}		}

4.3. Polymorphism and Dynamic Binding

Object references can have different types with different executions. That is, object references may refer to objects whose actual types differ from their declared types. The actual type can be of any type that is a subclass of the declared type. Polymorphism allows the behavior of an object reference to differ depending on the actual type. Therefore, it is important to identify and exercise the program with all possible type bindings. The polymorphism mutation operators are designed to ensure this type of testing.

• **PNC – *new* method call with child class type:** The PNC operator changes the instantiated type of an object reference. This causes the object reference to refer to an object of a type that is different from the declared type. In the example below, class *A* is the parent of class *B*.

Original Code		**PNC Mutant**
A a;		A a;
a = new A();	Δ	a = new B();

• **PMD – Member variable declaration with parent class type:** The PMD operator changes the declared type of an object reference to the parent of the original declared type. The instantiation will still be valid (it will still be a descendant of the new declared type). To kill this mutant, a test case must cause the behavior of the object to be incorrect with the new declared type. In the example below, class *A* is the parent of class *B*.

Original Code		**PMD Mutant**
B b;	Δ	A b;
b = new B();		b = new B();

• **PPD – Parameter variable declaration with child class type:** The PPD operator is the same as the PMD, except that it operates on parameters rather than instance and local variables. It changes the declared type of a parameter object reference to be that of the parent of its original declared type. In the example below, class *A* is the parent of class *B*.

Original Code		**PPD Mutant**
boolean equals (B o) { ... }	Δ	boolean equals (A o) { ... }

• **PRV – Reference assignment with other compatible type:** Object references can refer to objects of types that are descendants of its declared type. The ORR operator changes operands of a reference assignment to be assigned to objects of subclasses. In the example below, *obj* is of type *Object*, and in the original code it is given an object of type *String*. In the mutated code, it is given an object of type *Integer*.

Original Code		**OAC Mutant**
Object obj;		Object obj;
String s = "Hello";		String s = "Hello";
Integer i = new Integer(4);		Integer i = new Integer(4);
obj = s;	Δ	obj = i;

4.4. Method Overloading

Method overloading allows two or more methods of the same class to have the same name as long as they have different argument signatures. Just as with method overriding, it is important for testers to ensure that a method invocation invokes the correct method with appropriate parameters. Five mutation operators are defined to test various aspects of method overloading.

• **OMR – Overloading method contents change:** The OMR operator is designed to check that overloaded methods are invoked appropriately. The OMR operator replaces the body of a method with the body of another method that has the same name. This is accomplished by using the keyword *this*.

Original Code		**OMR Mutant**
class List {		class List{
... ...		... ...
void Add (int e) { ... ... }		void Add (int e) { ... ... }
void Add (int e, int n) {		void Add (int e, int n) {
... ...	Δ	this.Add(e);
}		}
}		}

• **OMD – Overloading method deletion:** The OMD operator deletes overloading method declarations, one at a

time in turn. If the mutant still works correctly without the deleted method, there may be an error in invoking one of the overloading methods; the incorrect method may be invoked or an incorrect parameter type conversion has occurred. The OMD operator ensures coverage of overloaded methods, that is, all the overloaded methods must be invoked at least once.

Original Code

```
class Stack extends List {
   ... ...
   void Push (int i) { ... }
   void Push (float i) { ... }
}
```

Δ

OMD Mutant

```
class Stack extends List {
   ... ...
   // void Push (int i) { ... }
   void Push (float i) { ... }
}
```

• **OAO – Argument order change:** The OAO operator changes the order of the arguments in method invocations, but only if there is an overloading method that can accept the new argument list. If there is one, the OAO operator causes a different method to be called, thus checking for a common fault in the use of overloading.

Original Code		**OAO Mutant**
s.Push (0.5, 2);	Δ	s.Push (2, 0.5);

• **OAN – Argument number change:** The OAN operator changes the number of the arguments in method invocations, but only if there is an overloading method that can accept the new argument list. Again, this helps ensure that the programmer did not invoke the wrong method. When new values need to be added, they are the constant default values of primitive types or the result of the default constructors for objects.

Original Code		**OAN Mutants**
s.Push (0.5, 2);	Δ	s.Push (2);
	Δ	s.Push (0.5);
	Δ	s.Push ();

4.5. Java-Specific Features

Java includes a few object-oriented language features that do not occur in all OO languages. This group of four operators attempt to ensure correct use of these features.

• **JTD –** *this* **keyword deletion:** The JTD operator deletes uses of the keyword *this*. Within a method body, uses of the keyword *this* refers to the current object. Typically, methods can refer directly to the object's instance variables by name. However, sometimes a member variable is hidden by a method parameter that has the same name, so *this* must be used. The JTD operator checks if the member variables are used correctly if they are hidden by a method parameters by replacing occurrences of "*this.X*" with "*X*" when "*X*" is both a parameter and an instance variable.

Original Code

```
class Stack {
   int size;
   ... ...
   void setSize (int size) {
      this.size=size;
   }
}
```

Δ

JTD Mutant

```
class Stack {
   int size;
   ... ...
   void setSize (int size) {
      size=size;
   }
}
```

• **JSC –** *static* **modifier change:** The JSC operator removes the *static* modifier to change class variables to instance variables, and it adds the *static* modifier to change instance variables to class variables. It is designed to validate behavior of instance and class variables.

Original Code		**JSC Mutant**
public static int s = 100;		public int s = 100;
private String name;	Δ	public static String name;

• **JID – Member variable initialization deletion**: Instance variables can be initialized in the variable declaration and in constructors for the class. The JID operator removes the initialization of member variables in the variable declaration so that member variables are initialized to the appropriate default values of Java. This is designed to ensure correct initializations of instance variables.

Original Code

```
class Stack {
   int size = 100;
   ... ...
   Stack() { ... ... }
}
```

Δ

JID Mutant

```
class Stack {
   int size;
   ... ...
   Stack() { ... ... }
}
```

• **JDC – Java-supported default constructor create**: Java creates *default* constructors if a class contains no constructors. The JDC operator forces Java to create a default constructor by deleting the implemented default constructor. It is designed to check if the user-defined default constructor is implemented properly.

Original Code

```
class Stack {
   ... ...
   Stack() { ... ... }
}
```

Δ

JDC Mutant

```
class Stack {
   ... ...
   // Stack() { ... ... }
}
```

4.6. Common Programming Mistakes

This category of mutation faults attempts to capture typical mistakes that programmers make when writing OO software. These are related to use of references and using methods to access instance variables.

• **EOA – Reference assignment and content assignment replacement:** Object references in Java are always through pointers. Although pointers in Java are typed, which is considered to help prevent certain types of faults, there are still mistakes that programmers can make. One common mistake is that of using an object reference instead of the contents of the object the pointer references. The EOA operator replaces an assignment of a pointer reference with a copy of the object, using the Java convention of a *clone()* method. The *clone()* method duplicates the contents of an object, creating and returning a reference to a new object.

Original Code		**EOA Mutant**
List list1, list2;		List list1, list2;
list1 = new List();		list1 = new List();
list2 = list1;	Δ	list2 = list1.**clone()**;

• **EOC – Reference comparison and content comparison replacement:** The EOC operator considers another common mistake with objects and object references. Comparisons of object references check whether the two references point to the same data object in memory. To support the comparison of the contents of objects, Java suggests the convention of an *equals()* method, which should take an object of type *Object* as a parameter and return a *boolean* value; true if the parameter has the same value as the reference object. This mutation operator targets faults programmers can easily make when confusing the reference of an object and its state.

Original Code		**EOC Mutant**
Fract f1 = new Fract (1, 2);		Fract f1 = new Fract (1, 2);
Fract f2 = new Fract (1, 2);		Fract f2 = new Fract (1, 2);
boolean b = (f1==f2);	Δ	boolean b = (f1.**equals** (f2));

• **EAM – Accessor method change:** Because private instance variables cannot be accessed outside of an object's own methods, good OO programming practice calls for providing public accessor and modifier methods by which clients of an object can effectively manipulate selected private instance variables. These are informally known as "get" and "set" methods, and by convention, use the variable name preceded by "get" or "set" (*getVariableName()*). One of the problems with this convention is that classes with multiple instance variables may wind up having many methods with very similar names. As a result, programmers easily get them confused.

The EAM operator changes an accessor method name for other compatible accessor method names, where *compatible* means that the signatures are the same except the method name. To kill this mutant a test case will have to produce incorrect output as a result of calling the wrong method.

Original Code		**EAM Mutant**
point.getX();	Δ	point.getY();

• **EMM – Modifier method change:** The EMM operator does the same as EAM, except it works with modifier methods ("set") instead of accessors.

Original Code		**EMM Mutant**
point.setX (2);	Δ	point.setY (2);

5. Summary and Comparison with Previous Mutation Operators

The operators in this paper are based on previous research that developed a fault model for object-oriented software. The goal was to develop mutation operators that can detect faults in this list. This section describes the relationships between the faults described in Section 3 and our mutation operators and the relationships between our operators and previous operators.

Table 2 relates the fault types and our mutation operators. All faults are covered, and some required multiple mutation operators. Conversely, some of the mutation operators cover more than one fault.

A number of the faults could be detected by previously defined mutation operators. Kim et al. [15, 16] defined 13 "class mutation operators". Two of Kim's operators mutate exception handling mechanisms, EHC and EHR. Although these are valid mutation operators, we do not consider exception handling to be strictly dependent on an object-oriented language, so do not consider them for our work. These operators should be used at the method testing level. Kim's POC operator changes the order of parameters, in effect, deleting a method and creating a new overloaded method. This new method is unlikely to be used during testing, which means the net result would be the same as the OMR mutation operator.

Chevalley [4] introduced three operators. All three of these should help detect faults from our list, so they have been incorporated into our list.

Table 3 relates our object-oriented mutation operators and the previously defined mutation operators; the previous operators prefixed with a "K-" are from Kim's list and those prefixed with a "C-" are from Chevalley's. When our mutation operators correspond to previous mutation operators, we use new names to ensure a standard naming convention. Several of our operators are similar to previous operators, but with some changes. The symbol * next to operator names indicates that there are some differences between ours and theirs. The CRT operator of Kim includes several types of changes, so it actually corresponds to several of our mutation operators. Chevalley's MNC operator

Table 2. Relation Between Faults and Operators

Faults	Class Mutation Operators
State visibility anomaly	IOP
State definition inconsistency (due to state variable hiding)	IHD, IHI
State definition anomaly (due to overriding)	IOD
Indirect inconsistent state definition	IOD
Anomalous construction behavior	IOR, IPC, PNC
Incomplete construction	JID, JDC
Inconsistent type use	PID, PNC, PPD, PRV
Overloading methods misuse	OMD, OAO, OAN
Access modifier misuse	AMC
static modifier misuse	JSC
Incorrect overloading methods implementation	OMR
super keyword misuse	ISK
this keyword misuse	JTD
Faults from common programming mistakes	EOA, EOC, EAM, EMM

applies to **all** methods, and it seems unlikely that programmers would make this mistake so generally. Thus, the MNC corresponds to our EAM and EMM, that is, restricting the changes to class accessor and modifier methods.

6. Cost and Future Work

One of the factors that has prevented mutation testing from being adopted for unit (method) level testing is that of cost. Mutation systems have been slow, requiring thousands of program executions for even small methods. The execution time is primarily a function of the number of mutants and various techniques have been proposed to reduce the number of mutants and to decrease the execution time per mutant. Mutation operator sets for procedural languages such as Fortran and C have led to $O(Var * Ref)$ mutants, where Var is the number of variables and Ref the number of references [19]. Selective mutation was able to reduce this to $O(Ref)$. In the following, we explore the cost of object-oriented mutation testing, then discuss construction of a mutation tool and future empirical work with the tool.

6.1. Number of Mutants

The number of mutants for each category of mutation operators is considered separately. Let V be the number of member variables for a class and M be the number of methods. Let RM be the number of overriding methods, CV be the number of object references of *polymorphic type*, that is, object references whose type can vary, and CR be the number of uses of polymorphic type objects. Let LM be the number of overloading methods, AM be the number of accessor and modifier methods, and CAM be the number of calls to accessor and modifier methods.

Information Hiding (Access Control). The number of mutants = $O(V + M)$.

Inheritance. Let S be the number of occurrences of the keyword *super*. The number of mutants = $O(V + RM + S)$.

Polymorphism and Dynamic Binding. The number of mutants is the number of object references whose type can vary dynamically times the number of uses of those object references. The number of mutants = $O(CV \times CR)$.

Overloading. Let CLM be the number of calls to an overloading method. The number of mutants = $O(CLM \times CV \times LM + LM^2)$.

Java-Specific. Let T be the number of occurrences of the keyword *this*. The number of mutants = $O(V + M + T)$.

Common Programmer Mistakes. The number of mutants = $O(AM \times CAM)$.

Which formula dominates depends on the characteristics of individual programs. Nevertheless, we suggest a simplification to a few dominating terms by the following analysis. It can safely be assumed that the keywords *super* and *this* are used rarely, thus terms involving Java-specific and inheritance mutation operators are unlikely to dominate. It is likely that the two largest values will be the number of instance variables and methods (V and M), however V and M are only involved in additive terms, so it is unlikely that they will dominate the number of mutants. If a program has many overloading methods and calls to them, the mutants from the overloading category may dominate. CLM and CV are almost certainly going to be small when compared with LM, thus the LM^2 term will dominate in this case.

Finally, we consider the mutation operators for common programmer mistakes. This number of accessors and modifiers (AM) will certainly vary depending on the program. However, accessors and modifiers account for most of the methods in many Java classes, thus this will be a large number and we expect this term to dominate ($O(AM \times$

Table 3. Mutation Operators for Inter-Class Testing

Operators	Description	Previous
AMC	Access modifier change	K-AMC
IHD	Hiding variable deletion	K-HFR
IHI	Hiding variable insertion	K-HFA
IOD	Overriding method deletion	K-OMR
IOP	Overridden method calling position change	
IOR	Overridden method rename	
ISK	*super* keyword deletion	
IPC	Explicit call of a parent's constructor deletion	
PNC	*new* method call with child class type	K-ICE
PMD	Instance variable declaration with parent class type	K-CRT *
PPD	Parameter variable declaration with child class type	K-CRT *
PRV	Reference assignment with other compatible type	
OMR	Overloading method contents change	
OMD	Overloading method deletion	K-VMR
OAO	Argument order change	K-AOC
OAN	Argument number change	K-AND
JTD	*this* keyword deletion	
JSC	*static* modifier change	K-SMC
JID	Member variable initialization deletion	
JDC	Java-supported default constructor create	
EOA	Reference assignment and content assignment replacement	C-RAC
EOC	Reference comparison and content comparison replacement	C-RCC
EAM	Accessor method change	C-MNC *
EMM	Modifier method change	C-MNC *

CAM))). Chevalley [5] found that over 30% of their mutants were of this category, lending some support to this theory.

Analytical predictions of the number of mutants has been historically difficult. Several formulae were claimed to represent the number of mutants for unit testing and the most commonly mentioned was (N^2) in the number of lines. A linear regression model on the Mothra Fortran mutation operators [19] found that the the actual number of mutants is $O(Vals*Refs)$, as claimed by Budd [3]. This history leads us to surmise that our preliminary analysis must be checked empirically before being accepted.

6.2. Mutation Tool

There are many aspects of generating mutation systems for class level testing of OO software that are more complicated than for procedural mutation systems. To apply many of the OO operators, we need to extract OO-specific information from the source code. For example, the IOD operator replaces an overriding method with the parent's overridden method. The mutation system needs to know the parent class and if the parent class has a method of the same name.

On the other hand, Java systems have a significant advantage – reflection. Java *reflection* is a technique for deriving name and signature information about classes directly from their bytecode, and then changing the behavior of a program according to another program called a meta-program [18, 26]. We developed our Java mutation tool following Chevalley's approach [5], which uses OpenJava [25], a compile-time reflective system. Figures 1 and 2 show the two main interfaces to this tool. Figure 1 shows the ability to generate the different types of mutants, and Figure 2 shows the ability to generate tests and run the mutants.

The reflection approach suffers from lack of efficiency; the source code is mutated so each mutant must be recompiled. We are currently developing a more powerful version of the tool using Javassist [6, 7]. Javassist is a load-time reflective system, and can directly mutate the bytecode. This significantly reduces the cost of mutation testing, and the new tool should be competitive with the cost of applying mutation using mutant schemata [24].

6.3. Effectiveness of the Operators

There are several issues that need to be considered to evaluate the usefulness and effectiveness of the OO class level mutation operators. First is the issue of equivalent mutants. Equivalent mutants do not affect the semantics of the program, therefore they are useless for mutation test-

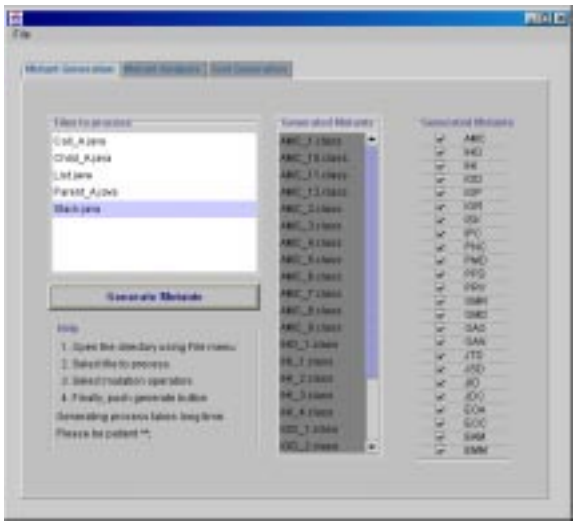

Figure 1. GUI for generating mutations

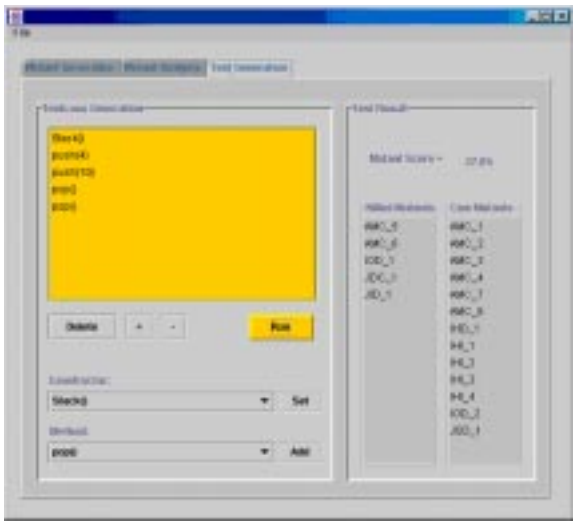

Figure 2. GUI for generating a test case and execution result

ing. However, the problem of detecting equivalent mutants is undecidable, and efforts to detect equivalent mutants have had mixed results [13, 21]. An upcoming experiment will help determine whether some operators generate too many equivalent mutants. Second, some operators can generate mutants that are easily killed. Although mutants make simple syntactic changes to the program, their semantic impacts can vary greatly. The impacts of OO operators on the semantics can vary from affecting the method to the semantics of the entire class. For example, the IOD operator swaps overriding methods with its parent's. The effect of IHD, on the other hand, extends over the whole class because it handles instance variable, which determine the class state. It is possible that some of these mutation operators will create mutants that are too easily killed. Finally, the mutation operators need to be evaluated in terms of their effectiveness of detecting faults in OO programs.

We are currently experimenting to evaluate the fault-finding effectiveness of the mutation operators. Initial results on small examples indicate that, as expected, the AMC operator creates the most mutants. Also, the AMC and JSC operators produced a lot of equivalent mutants, and the PNC, PMD, PPD and IHI operators produced equivalent mutants when overriding was present. Future work will elaborate on these very preliminary indications and should help us refine the set of mutation operators.

It is also possible that techniques such as weak mutation can be used [14, 20, 28]. However, a subtle point is that weak mutation is complicated by the information hiding that is inherent in OO software. Because internal data states are often less visible, weak mutation for OO software may well allow mutants to be killed much easier than for procedural software, which could lead to less effective tests.

6.4. Summary

This paper presents a comprehensive set of mutation operators to test for faults in the use of object-oriented features of Java. These mutation operators are based on an exhaustive list of OO faults, which gives them a firm theoretical basis. As a result, they correct several problems with the Java OO mutation operators that were previously published [4, 15, 16]. These mutation operators are designed with an emphasis on the integration aspects of Java to support **inter-class** level testing, and will help testers find faults with the use of language features such as access control, inheritance, polymorphism and overloading. Thus, this provides a way to improve the reliability of OO software.

The mutation operators are designed and expressed specifically for the Java language. This is necessary because mutation operators must take the semantics of a programming language into account. However, most of them can be easily adapted to other object-oriented programming languages, and we have tried to separate operators that are peculiar to Java into their own category.

The most essential theoretical component of a mutation system is that of the mutation operators, thus we have investigated this part of the problem first. However, this theory is not useful until it is followed up by the empirical work. We are currently beginning the experiment to investigate such questions as do these operators help find faults in OO software and should this set of operators be refined, reduced, or expanded. Past work [19] suggested that mutation should

try to only use operators that tend to produce mutants that have semantically small faults. Our future empirical work will have that as a goal.

References

[1] H. Agrawal, R. A. DeMillo, R. Hathaway, Wm. Hsu, Wynne Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. H. Spafford. Design of mutant operators for the C programming language. *Technical Report SERC-TR-41-P,Software Engineering Research Center, Purdue University*, March 1989.

[2] R. V. Binder. Testing object-oriented software: A survey. *Journal of Testing, Verification and Reliability*, 6(3/4):123–262, 1996.

[3] T. A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven CT, 1980.

[4] Philippe Chevalley. Applying mutation analysis for object-oriented programs using a reflective approach. In *Proceedings of the 8th Asia-Pacific Software Engineering Conference (APSEC 2001)*, Macau SAR, China, December 2001.

[5] Philippe Chevalley and Pascale Thevenod-Fosse. A mutation analysis tool for Java programs. *Journal on Software Tools for Technology Transfer (STTT)*, September 2001.

[6] S. Chiba. Javassist – A reflection-based programming wizard for Java. *Proceedings of OOPSLA '98 Workshop on Reflective Programming in C++ and Java*, October 1998.

[7] S. Chiba. Javassist WWW page. *http://www.csg.is.titech.ac.jp/ chiba/javassist/*, 2001.

[8] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface mutation: An approach to integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, March 2001.

[9] D.G. Firesmith. Testing object-oriented software. *In Proc. Technology of Object-Oriented Languages and Systems*, March 1993.

[10] L. Gallagher and A. J. Offutt. Integration testing of object-oriented components using finite state machines. *Subnmited for publication*, 2002.

[11] R. G. Hamlet. Testing programs with finite sets of data. *The Computer Journal*, 20(3):232–237, August 1977.

[12] Mary Jean Harrold and Gregg Rothermel. Performing data flow testing on classes. *Second ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE'94)*, pages 494–505, March 1994.

[13] R. M. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verication and Reliability*, 9(4):233–262, December 1999.

[14] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.

[15] S. Kim, J. Clark, and J. McDermid. Assessing test set adequacy for object oriented programs using class mutation. *28 JAIIO: Symposium on Software Technology*, Sept. 1999.

[16] S. Kim, J. Clark, and J. McDermid. Class mutation: Mutation testing for object-oriented programs. *OOSS: Object-Oriented Software Systems*, October 2000.

[17] R. J. Lipton, R. A. DeMillo, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[18] Sun Microsystems. Java reflection WWW page. *http://java.sun.com/j2se/1.4/docs/guide/reflection/index.html*, 2001.

[19] A. J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, April 1996.

[20] A. J. Offutt and S. D. Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, May 1994.

[21] A. J. Offutt and J. Pan. Detecting equivalent mutants and the feasible path problem. *The Journal of Software Testing, Verification, and Reliabili ty*, 7(3):165–192, September 1997.

[22] A. J. Offutt, J. Payne, and J. M. Voas. Mutation operators for Ada. Technical report ISSE-TR-96-09, Department of Information and Software Systems Engineering, George Mason University, Fairfax VA, March 1996. http://www.ise.gmu.edu/techrep/.

[23] Jeff Offutt, Roger Alexander, Ye Wu, Quansheng Xiao, and Chuck Hutchinson. A fault model for subtype inheritance and polymorphism. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, pages 84–93, Hong Kong China, November 2001. IEEE Computer Society Press.

[24] Jeff Offutt and Roland Untch. Mutation 2000: Uniting the orthogonal. In *Proceedings of Mutation 2000: Mutation Testing in the Twentie th and the Twenty First Centuries*, pages 45–55, San Jose, CA, October 2000.

[25] M. Tatsubori, S. Chiba, M.-O. Killijian, and K. Itano. OpenJava: A class-based macro system for Java. *Reflection and Software Engineering*, LNCS 1826:117–133, June 2000. Heidelberg, Germany.

[26] Paul Tremblett. Java reflection. *Dr. Dobb's Journal*, January 1998. http://www.ddj.com/articles/1998/9801/.

[27] Norman Wilde and Ross Huitt. Maintenance support for object-oriented programs. *Transactions on Software Engineering*, 18(12):1038–1044, Dec. 1992.

[28] M. R. Woodward and K. Halewood. From weak to strong, dead or alive? An analysis of some mutation testing issues. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 152–158, Banff Alberta, July 1988. IEEE Computer Society Press.