Description of Class Mutation Mutation Operators for Java

Yu-Seung Ma
Electronics and Telecommunications Research Institute, Korea
ysma@etri.re.kr

Jeff Offutt
Information and Software Engineering
George Mason University
offutt@ise.gmu.edu

November 7, 2005

This document provides a brief description of the muJava class mutation operators, which were updated currently for version II of the tool. The class mutation operators are classified into four groups, based on the language features that are affected. The first three groups are based on language features that are common to all OO languages. The last group includes OO features that are Java-specific.

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Java-Specific Features

Our strategy for developing mutation operators is to handle all the possible syntactic changes for OO features. Some previous mutation operators have been developed based on experience of testers. There is so little reported data on testing OO features that this method is not possible at this time.

Generally, all the behaviors of mutation operators fall under one of the three categories: (1) **delete**, (2) **insert**, and (3) **change** a target syntactic element. This paper reports on mutation operators that implement all three kinds of behaviors, within Java's syntactic rules.

This section first describes each operator informally, then gives an example mutant that can be created from the operator. Several operators were taken from the previous research into OO mutation, and the relationships between our operators and the previous operators are detailed in the next section.

Language Feature	Operator	Description
Encapsulation	AMC	**A**ccess modifier **c**hange
Inheritance	IHD	**H**iding variable **d**eletion
	IHI	**H**iding variable **i**nsertion
	IOD	**O**verriding method **d**eletion
	IOP	**O**verriding method calling **p**osition change
	IOR	**O**verriding method **r**ename
	ISI	`super` keyword **i**nsertion
	ISD	`super` keyword **d**eletion
	IPC	Explicit call to a **p**arent's **c**onstructor deletion
Polymorphism	PNC	**n**ew method call with **c**hild class type
	PMD	**M**ember variable **d**eclaration with parent class type
	PPD	**P**arameter variable **d**eclaration with child class type
	PCI	Type **c**ast operator insertion
	PCC	**C**ast type **c**hange
	PCD	Type **c**ast operator **d**eletion
	PRV	**R**eference assignment with other comparable **v**ariable
	OMR	**O**verloading **m**ethod contents **r**eplace
	OMD	**O**verloading **m**ethod **d**eletion
	OAC	**A**rguments of **o**verloading method call **c**hange
Java-Specific Features	JTI	`this` keyword **i**nsertion
	JTD	`this` keyword **d**eletion
	JSI	`static` modifier **i**nsertion
	JSD	`static` modifier **d**eletion
	JID	Member variable **i**nitialization **d**eletion
	JDC	Java-supported **d**efault constructor **c**reation
	EOA	**R**eference **a**ssignment and content assignment replacement
	EOC	**R**eference **c**omparison and content comparison replacement
	EAM	**A**cessor **m**ethod change
	EMM	**M**odifier **m**ethod change

Table 1: Mutation Operators for Inter-Class Testing

1 Encapsulation

In our experience in teaching OO software development and consulting with companies that rely on OO software, we have observed that 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.

- **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.

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

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 shadowing, method overriding, the use of `super`, and definition of constructors.

Variable shadowing can cause instance variables that are defined in a subclass to shadow (or hide) member variables of the parent. However, 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 shadowing is used, which is the intent of the IHD and IHI mutation operators.

- **IHD – Hiding variable deletion:** The IHD operator deletes a hiding variable, a variable in a subclass that has the same name and type as a variable in the parent class. This causes references to that variable to access the variable defined in the parent (or ancestor). 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 a hiding variable into a subclass. It is a reverse case of IHD. By inserting a hiding variable, two variables (a hiding variable and a hidden variable) of the same name become to be exist. Newly defined and overriding methods in a subclass reference the hiding variable although inherited methods reference the hidden variable as before.

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

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 are overriding methods, it is important for testers to ensure that a method invocation actually invokes the intended method.

- **IOD – Overriding method deletion:** 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. The mutant act as if there is no overriding method for the method.

Original Code		**IOD Mutant**
class Stack extends List {		class Stack extends List {
... ...		... ...
void push (int a) { ... }	Δ	// void 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;
    }
}
```

IOP Mutant

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

class Stack extends List {
    ... ...
    void SetEnv() {
Δ       size = 10;
Δ       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. It models the situation that the overriding method is declared as a new method with different name in the child class.

Original Code

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

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

IOR Mutant

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

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

The `super` keyword is used to access parent's members (variables or methods) within the child class. When there is variable shadowing or method overriding, use of the `super` keyword should be careful because it change the reference to variables or methods from the child class to super class.

- **ISI – `super` keyword insertion:** The ISI operator inserts the `super` keyword so that a reference to the variable or the method goes to the overridden instance variable or method. The ISI 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*num;
    }
}
```

Δ

ISK Mutant

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

- **ISD – `super` keyword deletion:** The ISD 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. It is a reverse case of the ISI.

Original Code

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

Δ

ISK Mutant

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

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.

- **IPC – Explicit call of a parent's constructor deletion:**

  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

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

Δ

IPC Mutant

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

3 Polymorphism

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 from any type that is a subclass of the declared type. Polymorphism allows the behavior of an object reference to be different depending 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 POI 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 `Parent` is the parent of class `Child`.

Original Code		**PNC Mutant**
Parent a;		Parent a;
a = new Parent();	Δ	a = new Child();

- **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 `Parent` is the parent of class `Child`.

Original Code		**PMD Mutant**
Child b;	Δ	Parent b;
b = new Child();		b = new Child();

- **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 `Parent` is the parent of class `Child`.

Original Code		**PPD Mutant**
boolean equals (Child o) { ... }	Δ	boolean equals (Parent o) { ... }

- **PCI – Type cast operator insertion:** The PCI operator changes the actual type of an object reference to the parent or child of the original declared type. The mutant shows different behavior when the object to be casted has hiding variables or overriding methods.

Original Code		**PCI Mutant**
Child cRef;		Child cRef;
Parent pRef = cRef;		Parent pRef = cRef;
pRef.toString();	Δ	((Child)pRef).toString();

- **PCD – Type cast operator deletion:** The PCD operator deletes type casting operator. It models a reverse case of PCI.

Original Code		**PPD Mutant**
Child cRef;		Child cRef;
Parent pRef = cRef;		Parent pRef = cRef;
((Child)pRef).toString();	Δ	pRef.toString();

- **PCC – Cast type change:** The PCC operator change the type that a variable is to be cast into. The change is occurred with subclasses or ancestors of the type.

Original Code		**PPD Mutant**
((Parent)ref).toString();	Δ	((Child)ref).toString();

- **PRV – Reference assignment with other compatible type:** Object references can refer to objects of types that are descendants of its declared type. The PRV 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		**PRV 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;

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. Three 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

```
class List {
    ... ...
    void Add (int e) { ... ... }
    void Add (int e, int n) {
        ... ...
    }
}
```

Δ

OMR Mutant

```
class List{
    ... ...
    void Add (int e) { ... ... }
    void Add (int e, int n) {
        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 POD 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) { ... }
}
```

- **OAC – Argument of overloading method change:** The OAC operator changes the order or the number 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 OAC operator causes a different method to be called, thus checking for a common fault in the use of overloading.

Original Code

```
s.Push (0.5, 2);
```

OAC Mutants

Δ `s.Push (2, 0.5);`
Δ `s.Push (2);`
Δ `s.Push (0.5);`
Δ `s.Push ();`

4 Java-Specific Features

Some object-oriented features are not common to all object-oriented languages. This group of operators attempt to ensure correct use of such features supported in Java. Also, mistakes that a programmer often makes when writing object-oriented software are considered here.

- **JTI – `this` keyword insertion:** The JTI operator inserts the keyword `this`. 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) {
        this.size=this.size;
    }
}
```

- **JTD – `this` keyword deletion:** The JTD operator deletes uses of the keyword `this`. It models a reverse case of JTI.

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;
    }
}
```

- **JSI – `static` modifier insertion:** The JSI operator adds the `static` modifier to change instance variables to class variables. It is designed to validate behavior of instance and class variables.

Original Code

```
public int s = 100;
```

Δ

JSC Mutant

```
public static int s = 100;
```

- **JSD – `static` modifier deletion:** The JSD operator removes the `static` modifier to change class variables to instance variables. It models a reverse case of JSI.

Original Code

```
public static int s = 100;
```

Δ

JSC Mutant

```
public int s = 100;
```

- **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() { ... ... }
}
```

- **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**
Stack s1, s2;		Stack s1, s2;
s1 = new Stack();		s1 = new Stack();
s2 = s1;	Δ	s2 = s1.**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 `java.lang.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**
Integer i1 = new Integer (7);		Integer i1 = new Integer (7);
Integer i2 = new Integer (7);		Integer i2 = new Integer (7);
boolean b = (i1==i2);	Δ	boolean b = (i1.**equals** (i2));

- **EAM – Accessor method change:**

  The EAM operator changes an accessor method name for other compatible accessor method names, where *compatible* means that the signatures are the same. This type of mistake occurs because classes with multiple instance variables may wind up having many accessor methods with the same signature and very similar names. As a result, programmers easily get them confused. 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 EMM, except it works with modifier methods instead of accessor methods.

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