MuJava : An Automated Class Mutation System

Yu-Seung Ma[1]*, Jeff Offutt[2]†, and Yong Rae Kwon[1]

[1]Division of Computer Science
Department of Electrical Engineering and Computer Science
Korea Advanced Institute of Science and Technology, Korea
{*ysma, kwon*}*@salmosa.kaist.ac.kr*

[2]Department of Information and Software Engineering
George Mason University
ofut@ise.gmu.edu

Abstract

Several module and class testing techniques have been applied to object-oriented programs, but researchers have only recently begun developing test criteria that evaluate the use of key OO features such as inheritance, polymorphism, and encapsulation. Mutation testing is a powerful testing technique for generating software tests and evaluating the quality of software. However, the cost of mutation testing has traditionally been so high it cannot be applied without full automated tool support.

This paper presents a method to reduce the execution cost of mutation testing for OO programs by using two key technologies, Mutant Schemata Generation (MSG) and bytecode translation. This method adapts the existing MSG method for mutants that change the program behavior and uses bytecode translation for mutants that change the program structure. A key advantage is in performance: only two compilations are required and both the compilation and execution time for each is greatly reduced.

A mutation tool based on the MSG/bytecode translation method has been built and used to measure the speedup over the separate compilation approach. Experimental results show that the MSG/bytecode translation method is about five times faster than separate compilation.

Keywords

Object-Oriented Programs, Mutation Testing, Software Testing

1 Introduction

Object-oriented design, programming, and languages offer many advantages to software developers and provide solutions to old problems. However, the novel object-oriented language features introduce new kinds of problems that in some cases require novel solutions [6]. Researchers have been developing new methods and techniques to test object-oriented software for a number of years. Early work focused on testing of data abstractions and state behavior [7, 20, 21, 24, 31, 40, 55, 58]. Subsequent work looked into testing of classes and issues such as what kind and how many objects should be instantiated and in what order classes should be tested [8, 9, 26, 39]. More recently, researchers have looked at integration issues of OO software and testing of complete classes [23, 27, 28, 54].

* *Supported in part by the Korean Science and Engineering Foundation under grant GH15420 to KAIST.*
† *Supported in part by the U.S. National Science Foundation under grant CCR-98-04111 to George Mason University.*

Finally, researchers have started to look for ways to test the essential OO language features of inheritance and polymorphism [4, 43, 54]. Several different ideas have been put forward, each with advantages and disadvantages. One recent idea is to use mutation testing to test classes; approaches have been proposed by Kim, Clark and McDermid [34, 35], Chevalley and Thévenod-Fosse [10, 11], Ma, Kwon and Offutt [43], and Alexander et al. [3].

Mutation testing [25, 41] 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 a majority of simple faults are detected and removed. Simple faults are introduced into the program by creating a set of faulty versions, called *mutants*. These mutants are created from the original program by applying *mutation operators*, which describe syntactic changes to the programming language. 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 one or more mutants is considered to be *effective* at finding faults in the program.

Mutation testing involves many executions of programs; thus cost has always been a serious issue. Many techniques for implementing mutation testing have proved to be too slow for practical adoption. This paper presents a design and results from an implementation of a mutation system that is based on a novel execution strategy that combines mutation schemata [60] with bytecode translation.

1.1 Mutation Execution Speed

Several approaches have been developed to reduce the computational expense of the mutation testing. Untch categorized the approaches into three strategies, *do fewer*, *do smarter*, and *do faster* [59, 60]. The *do fewer* approaches try to run fewer mutant programs without incurring intolerable loss in effectiveness. The *do smarter* approaches seek to distribute the computational expense over several machines or factor the expense over several executions by retaining state information between runs. The *do faster* approaches focus on ways to generate and run mutant programs as quickly as possible. These methods have been developed for traditional programming languages, and are not all applicable to OO languages.

This paper presents a *do faster* method for OO inter-class mutation testing. This involves examining whether existing *do faster* methods can be applied to object-oriented programs. This approach primarily attempts to reduce the compilation time. These ideas have been implemented in an automated OO mutation system, which has been compared with previous execution techniques. Most of the OO mutation operators are independent of language; however, they have only been implemented in Java and so have some Java dependencies. The implementation method depends on the use of reflection, so can only be used in languages that support reflection.

The contents of the paper is as follows. Section 2 summarizes the current status of OO mutation testing. Section 3 describes the existing cost reduction methods for mutation testing. Section 4, the main part of the paper, presents the technique for executing mutation on OO programs and describes the tool. Section 5 presents experimental results from a cost comparison. Section 6 presents conclusions and discusses future work.

2 Mutation Testing for OO Programs

OO programs have many characteristics that differ from traditional programs. They are often structured differently and they contain new features such as encapsulation, inheritance, and polymorphism. These differences and new features in OO programs change the requirements for mutation testing.

A major difference for testers is that OO software changes the levels at which testing is performed [23, 27]. In OO software, unit and integration level testing can be classified into four levels: (1) *intra-method*, (2) *inter-method*, (3) *intra-class*, and (4) *inter-class*. This classification follows definitions by Harrold and Rothermel [27] and Gallagher and Offutt [23].

- *Intra-method Level:*
  Intra-method level faults occur when the functionality of a method is implemented incorrectly. Testing

within classes corresponds to unit testing in conventional programs. So far, researchers have assumed that traditional mutation operators for procedural programs [2, 16, 51] will suffice for this level (with minor modifications to adapt to new languages).

- *Inter-method Level:*
  Inter-method level faults are made on the connections between pairs of methods of a single class. Testing at this level is equivalent to integration testing of procedures in procedural language programs. Interface mutation [16], which evaluates how well the interactions between various units have been tested, is applicable to this level.

- *Intra-class Level:*
  Intra-class testing is when tests are constructed for a single class, with the purpose of testing the class as a whole. Intra-class testing is a specialization of the traditional unit and module testing. It tests the interactions of public methods of the class when they are called in various sequences. Tests are usually sequences of calls to methods within the class, and include thorough tests of public interfaces to the class.

- *Inter-class Level:*
  Inter-class testing is when more than one class is tested in combination to look for faults in how they are integrated. Inter-class testing specializes the traditional integration testing and seldom used subsystem testing, where most faults related to polymorphism, inheritance, and access are found.

Another difference for object-oriented languages is that mutation operators that handle new types of faults [53] introduced by OO-specific features are needed. A set of class mutation operators [34, 35, 43] has been developed for this purpose.

Third, an OO mutation system should be able to extract information and execute programs from an object-oriented standpoint. For example, user-defined types (that is, classes) and references to user-defined types must be handled. Control, data, inheritance and polymorphic relationships among components should also be considered. The following subsections briefly summarize the previous research in OO mutation testing. Most of this has focused on developing mutation operators rather than developing algorithms and techniques for implementing them in usable and efficient tools.

The tool described in this paper implements both inter- and intra-class mutation operators. This paper primarily focuses on studying the inter-class, or object-oriented, operators. Much is already known about intra-class (statement level) mutation operators.

2.1 Class Mutation Operators

The first attempt to define mutation operators to detect faults related to OO-specific features was by Kim et al. [35]. They designed thirteen class mutation operators that were extended to sixteen by Chevalley [10]. A subsequent systematic classification of OO specific faults in terms of language syntax by Offutt et al. [53] revealed several types of OO faults that the previous operators do not model. Alexander et al. [3] proposed a different scheme for mutating objects. Their approach relies on making changes to the data state of objects during execution rather than the program.

Based on this fault classification and previous work, Ma et al. [43] developed a comprehensive set of class mutation operators for Java. This set of 24 mutation operators is summarized in Table 1. Each mutation operator is related to one of the following six language feature groups. 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. Complete and precise definitions of these operators were presented in a previous paper [43]. As is usual with mutation operators, they are only applied in situations where the mutated program will still compile.

1. **Information Hiding (Access Control)**
   In our experience, access control is a common source of mistakes among OO programmers. The

Language Feature	Operator	Description
Access Control	AMC	Access modifier change
Inheritance	IHD	Hiding variable deletion
	IHI	Hiding variable insertion
	IOD	Overriding method deletion
	IOP	overriding method calling position change
	IOR	Overriding method rename
	ISK	*super* keyword deletion
	IPC	Explicit call of a parent's constructor deletion
Polymorphism	PNC	*new* method call with child class type
	PMD	Instance variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PRV	Reference assignment with other comparable type
Overloading	OMR	Overloading method contents change
	OMD	Overloading method deletion
	OAO	Argument order change
	OAN	Argument number change
Java-Specific Features	JTD	*this* keyword deletion
	JSC	*static* modifier change
	JID	Member variable initialization deletion
	JDC	Java-supported default constructor creation
Common Programming Mistakes	EOA	Reference assignment and content assignment replacement
	EOC	Reference comparison and content comparison replacement
	EAM	Accessor method change
	EMM	Modifier method change

Table 1: Mutation Operators for Inter-Class Testing

semantics of the various access levels are often poorly understood, and access for variables and methods is not always considered during design. 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. The AMC mutation operator has been developed for this category.

2. **Inheritance**
   Although inheritance is a powerful and useful abstraction mechanism, incorrect use can lead to a number of faults. Seven mutation operators have been defined to test the various aspects of using inheritance, covering variable hiding, method overriding, the use of *super*, and definition of *constructor*s.

3. **Polymorphism**
   Polymorphism and dynamic binding allow object references to take on different types in different executions and at different times in the same execution. That is, object references may refer to objects whose actual types differ from their declared types. In most languages (including Java and C++), the actual type can be 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. Four operators have been developed for this category.

4. **Overloading**
   Method overloading allows two or more methods of the same class or type family to have the same name as long as they have different argument signatures. Just as with method overriding (polymorphism), it is important for testers to ensure that a method invocation invokes the correct method with appropriate parameters. Four mutation operators have been defined to test various aspects of method overloading.

5. **Java-Specific Features**
   Because mutation testing is language dependent, mutation operators need to reflect language-specific features. Java has a few object-oriented language features that do not occur in all OO languages and four operators have been defined to ensure correct use of these features. They cover use of *this*, *static*, default constructors and initialization.

6. **Common Programming Mistakes**
   This category 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. Four operators have been developed for this category.

2.2 Mutation Tools for OO Testing

Mutants are created from a test program using an automated mutation system. Test cases are then added in an attempt to "kill" the mutants by differentiating the output of the original program from the mutant programs.

Mutation tools must be designed and implemented differently for OO languages. Mutation operators for conventional (non-OO) languages do not change type or data structure declarations. The mutation operators that have been designed for Java do, so an OO mutation system needs to be able to make those changes. It must also access information in a program from an OO standpoint. For instance, the CRT operator, which replaces reference types with compatible types, needs to use inheritance relationships to create mutants. Also, abstract syntax trees, which are typically used in procedural mutation systems, do not allow immediate access to object-oriented features of the source code.

One of the first implementations for OO mutation was by Chevalley and Thévenod-Fosse [10]. They pointed out that the traditional syntactic analysis technique is not sufficient for implementing an OO mutation system and suggested using *reflection* [33, 44] to overcome this problem.

Reflection [33, 44] allows a program to (1) access its internal structure and behavior, and (2) manipulate that structure, thereby modifying its behavior based on rules supplied by another program. *Reflection* is a natural way to implement mutation analysis for several reasons. First, it lets programmers extract OO-related information about a class by providing an object that represents a logical structure of the class definition. This helps solve the first problem in implementing a mutation analysis system, parsing the program. Second, it provides an API to easily change the behavior of a program during execution. This can be used to create mutated versions of the program. Third, it allows objects to be instantiated and methods to be invoked dynamically. Java provides a built-in reflection capability with a dedicated API [47]. This allows Java programs to perform functions such as asking for the class of a given object, finding the methods in that class, and invoking those methods. However, the Java language as defined does not provide full reflective capabilities. Specifically, Java only supports introspection, which is the ability to introspect data structures, but does not support alteration of the program behavior. Several reflection systems [12, 36, 57, 61] have been proposed to complement the Java reflection API [47]. Because of differences in these systems, which reflection system is selected can significantly affect the efficiency of mutation analysis and testing.

Chevalley and Thévenod-Fosse's tool established the feasibility of applying mutation analysis to class-level testing and identified three problems. First, the research was developed before the class mutation operators were well established and did not support all the current mutation operators. Second, their tool separately compiles each mutated class, a very slow process. Finally, their tool generates mutants but does not support the execution of mutants and tests. The research project reported here started with Chevalley and Thévenod-Fosse's work and has the goal of addressing the remaining problems.

A previous paper [43] reported on mutation operators that more completely addresses potential OO faults. This paper reports on algorithms and techniques for efficient implementation of a full class-level mutation system. A followup paper [42] reported on a mutation tool that supports the entire process of mutation testing.

However, these two approaches [10, 42] have a very serious drawback – low performance. They are slow because they use an inefficient way to create mutant programs: creating a copy of the original source code

and then changing it for each mutant. Although this approach is very simple, it is too computationally and spatially expensive because hundreds of mutated programs must be stored and compiled. Therefore, a more efficient method to improve the performance is needed to allow useful tools to be built.

To solve this problem, this paper proposes a novel approach that generates mutants directly from the bytecode. Since the approach works directly on the bytecode, it is not necessary to recompile each mutant, a serious performance problem with previous mutation tools. The following section briefly describes bytecode translation toolkits. The paper then discusses a way to improve the performance of mutation testing, and follows with results from a proof-of-concept implementation.

2.3 Bytecode Translation

Bytecode translation is a technique that has some abilities in common with reflection, but uses a very different approach. Bytecode translation inspects and modifies the intermediate representation of Java programs, bytecode. Since it directly handles bytecode, it has some advantages over source-level translation. First, it can process an off-the-shelf program or library that is supplied without source code. Second, it can be performed on demand at load time, when the Java virtual machine loads a class file.

Several toolkits are available for bytecode translation, including BCA [32], BCEL [15], Javassist [12], JMangler [37], and JOIE [14]. This section presents some of the abilities of bytecode translation and discusses how some of these tools implement them. The reader should be aware that this is **not** a complete survey or comprehensive overview of the technique; the goal of this section is to provide enough background so that the reader can understand how bytecode translation is used in this research.

1. **Kinds of Translations**
   BCA [32], Javassist [12], and JMangler [37] place restrictions on the kinds of modifications that transformers are allowed to make. The intent is to make it less likely that a change will result in a program that would not have passed a compiler check and that will correctly run in the JVM. For example, Javassist [12] does not allow existing fields and methods to be removed. However, BCEL [15] and JOIE [14] allows all kinds of modifications to be made, including addition, change and removal of fields and members, as well as arbitrary changes to code.

2. **JVM Independence**
   BCA [32] requires a modified JVM to work with the class loading process. Therefore, programmers must use a specific platform and a specific JVM. The other bytecode translators are implemented in pure Java, so are independent of the JVM.

3. **Correctness**
   All the transformed bytecodes should be structurally correct, and also be compatible with other binaries that the classes are linked with. Otherwise the transformed class may not work correctly. Javassist [12] and JMangler [37] enforce this restriction, but other bytecode translators do not.

4. **Ease of Use**
   Bytecode translators can be difficult to use for developers who are not intimately familiar with Java bytecode. BCA [32] and Javassist [12] were designed to be easy to learn with only a limited knowledge of bytecode, and our experience has found this to be true. However, both BCA and Javassist cover a limited set of modifications.

3 Overview of a Java OO Mutation System

This paper presents the design of and results from an OO mutation testing system that uses MSG [60] and bytecode translation. It uses MSG to generate one "meta-mutant" program at the source level that incorporates many mutants. The tool, *MuJava*, works directly on the bytecode; thus it only requires two compilations: compilation of the original source code and compilation of the metamutants generated with

MSG. This design allows faster performance than mutation systems that compile all mutants such as the two previous OO Java systems [10, 42] and Proteum for C [16, 17]. Older tools such as Mothra [19] modified and interpreted intermediate code that was designed specifically for mutation. Since interpretation generally runs around 10 times slower than compiled programs, MSG and bytecode translation can be significantly faster.

The class mutation operators have different characteristics from traditional mutation operators. In particular, they require changes to the structure of the program such as definitions of class variables. Traditional mutation operators only change the behavior of the program, not the structure. Therefore new methods are needed to change the structure.

This section describes how to deal with these problems. First, the *Mutant Schemata Generation* (MSG) method [60], is adapted for class mutation operators that change the behavior of the program. It creates a "meta" version of the test program that contains all mutants and requires only one compilation. A new method based on bytecode translation is introduced for class mutation operators that change the structure of the program. Since bytecode translation allows the structure of the bytecode to be changed directly, it does not require additional compilation.

Mutants that change the behavior of the program are called *behavioral mutants* and the mutants that change the structure of the program are called *structural mutants*. The class mutation operators from Table 1 fall into these categories as follows:

- **Behavioral Class Mutation Operators**
  IOP, ISK, PNC, PRV, OMR, OAO, OAN, JTD, EOA, EOC, EAM, EMM
- **Structural Class Mutation Operators**
  AMC, IHD, IHI, IOD, IOR, IPC, PMD, PPD, OMD, JSC, JID, JDC

Note that the experimental results in this paper do not use the AMC operator. AMC changes the access level, which in the vast majority of the cases either creates a mutant program that will not compile (breaking the access) or creates a mutant program that is equivalent to the original program (wider access). The only situation where AMC can create a killable mutant is when it creates a naming conflict that is resolved dynamically, and the mutant program is resolved differently from the original. This is such a rare case that we feel the AMC operator is not useful, and include it in the discussion only because it appeared in the previous conference paper [43]. Also, we merged OAO into OAN because the behaviors of these two operators are similar. So, all the results from the OAN operator of this paper corresponds to the combined results from both OAO and OAN operators.

Figure 1 illustrates the overall structure of MuJava. Behavioral mutants and structural mutants are generated and executed by different engines, then their results are combined. The following subsections describe the major steps summarized in this figure.

3.1 Generating and Running Behavioral Mutants

Traditional (non-OO) mutants are all behavioral in nature, so existing mutant generation techniques can be used. This paper adapts the MSG method [60], which has been found to be significantly faster than interpretive systems for traditional intra-class mutation operators. MuJava implements both inter- and intra-class mutation operators, but this research primarily focuses on studying inter-class mutation. The following subsections briefly describe MSG and how to use it to generate and execute behavioral mutants.

3.1.1 MSG Method

The MSG method encodes all mutants for a program into a specially parameterized program, called a *metamutant*. The metamutant is derived from the program under test P, the metamutant is compiled using the same standard compiler used to compile P, and it runs at compile-speeds. While running, the metamutant has the ability to function as any of the mutant programs of P.

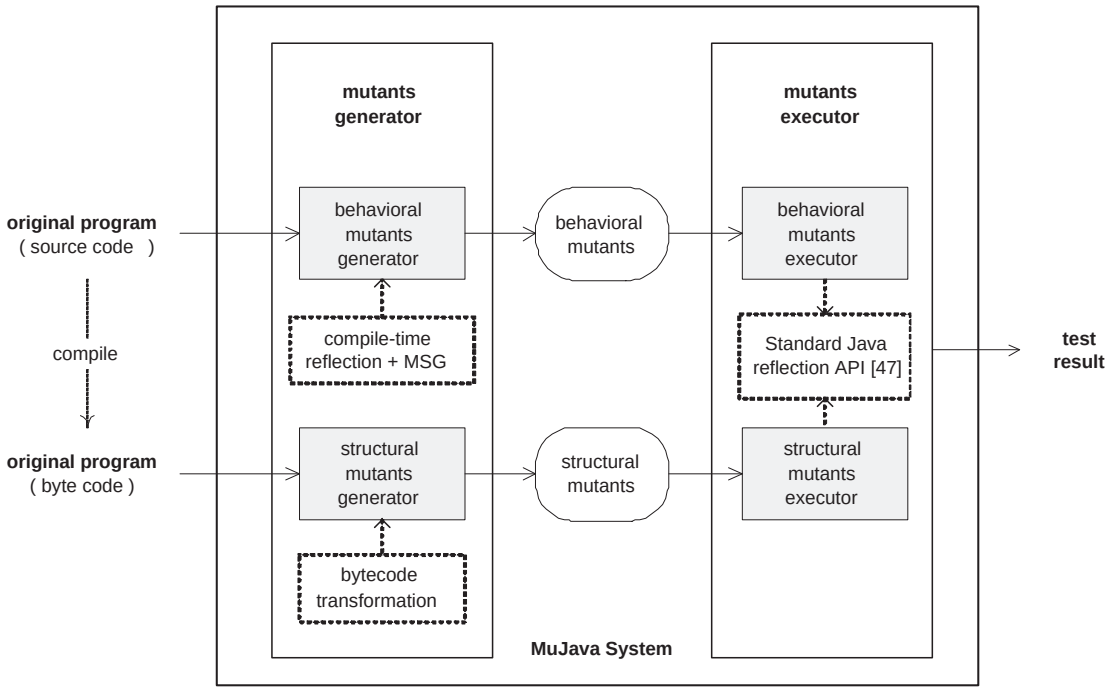

Figure 1: **Overall Structure of MuJava**

To explain how one metamutant can represent all the functionality of an entire collection of mutants, mutation analysis must be understood. Recall that for a program P, each mutant of P is formed as a result of a single modification to some statement in P. Thus, each mutant of P differs from the original by only one mutated statement. The way in which these statements are altered is dictated by the set of mutation operators used.

Consider the *arithmetic operator replacement (AOR)* operator (AOR is not an OO mutation operator; it is used to provide a simple example). The AOR operator replaces each occurrence of an arithmetic operator by each of the other possible arithmetic operators. Applying this rule to the assignment statement *Result = A - B* yields the following four mutated statements:

Result = A + B;
Result = A * B;
Result = A / B;
Result = A % B;

These mutations can be generically represented as

Result = A *ArithOp* B;

where *ArithOp* is a *metaoperator* abstract entity. The generic representation above can be rewritten as the syntactically valid statement

Result = AOrr (A, B);

where the AOrr function performs one of the five possible arithmetic operations. AOrr is an example of a

metaprocedure, a function that corresponds to an abstract entity in the schema. A statement that has been changed to this type of generic form is said to have been *metamutated*. A *metamutation* is a syntactically valid change that embodies other changes.

When generating the metamutant of a program P, a list of mutant descriptors is produced. This list details the alternate operations to be used at each change point in the program. Using this list, the metamutant is dynamically instantiated to function as any of the mutants of P.

3.1.2 Implementation Details

The process of generating and running behavioral mutants is illustrated in Figure 2. Major components from Figure 1 are elaborated inside the *behavioral mutants generator* and *behavioral mutants executor* boxes. Compile-time reflection [29] is used to analyze the original program source and to generate Java source for mutants. MuJava uses OpenJava [56, 57] for compile-time reflection because it provides enough information to generate mutants and it is easy to use. The MSG engine uses compile-time reflection to generate a metamutant program that has a metaprocedure call and definitions of metaprocedures. The metamutant is generated as source, then compiled. To execute mutants, the metamutant is loaded into the JVM, then mutants are obtained by using the list of mutant descriptors to repeatedly instantiate the metamutant.

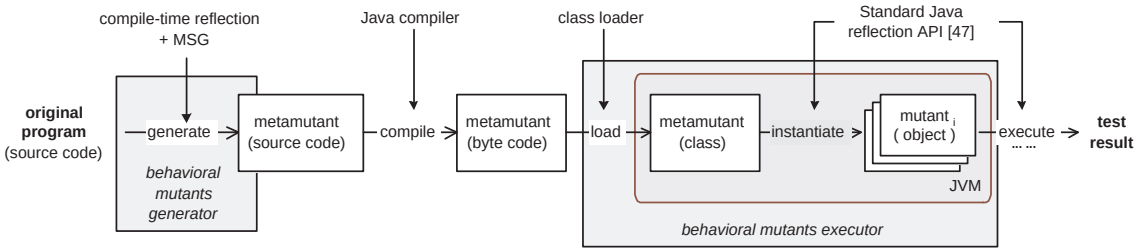

Figure 2: **Mutation Process for Behavioral Mutants**

Four additional issues must be considered before using the MSG method for OO programming languages: (I) object references, (II) polymorphism, (III) instantiation overhead, and (IV) the number of mutants. These issues are illustrated below in the context of specific mutation operators; other operators are similar in nature.

I. Object

An object is a dynamically created instance of a class pointed to by references. Unlike traditional programs, an OO program is executed by a set of objects. Most variable accesses and method calls are accomplished by referring to an object. Therefore, the metaprocedure must be modified to accept an object reference as an additional parameter.

Consider the OAN mutation operator that changes the number of the arguments in overloading method invocations. Assume that there is a variable *a* of type *A* and class *A* has two overloaded methods with the signatures *void f (int, int, char)* and *void f (int, char)*. Then, for the statement below,

```
a.f (2, 3, 'c');
```

two mutated forms, *a.f(2,'c')* and *a.f(3,'c')*, are syntactically possible. Mutation creates all mutants that are syntactically correct. So, the tool does standard compile-time type checking based on inheritance information and Java type conversion rules. MSG changes the original statement to a metaprocedure *f_oan (a, 2, 3, 'c');*. Note that an additional argument, *a*, is added to pass the object reference as a parameter. The implementation of *f_oan()* is given as.

```
void f_oan (A obj, int p1, int p2, char p3)
```

```
{
    switch (mutantID)
    {
        case 1:  obj.f (p1, p3);   break;  // a.f (2, 'c');
        case 2:  obj.f (p2, p3);   break;  // a.f (3, 'c');
        default: obj.f (p1, p2, p3);       // original code
}
```

The object reference is needed to call the method, so the parameter *obj* of type A is included in the parameter list of the metaprocedure *f_oan()*. The metaprocedure *f_oan()* generates two mutants: *a.f (2, 'c')* and *a.f (3, 'c')* according to the value of *mutantID*.

II. Polymorphism

Polymorphism allows an object reference to take on any type that is a descendant of the declared type. This presents difficulties for mutation because the operators for polymorphism generate mutants of diverse types.

The PNC mutation operator causes the object reference to refer to different types from the declared type. If class B inherits from class A and C inherits from B, for the statement below,

```
A a = new A();
```

a can reference an object of type B or C. The PNC operator generates two mutated statements (PNC is only applied when the constructor signatures match):

```
A a = new B();      // mutated statement 1
A a = new C();      // mutated statement 2
```

A return type must be chosen to design the metaprocedure for this example. A metaprocedure should be able to return instances of type A, B, and C.

Java allows automatic object reference conversion when the conversion is "up" the inheritance hierarchy. Therefore, for the statement *A a = new B();*, the reference of type B is automatically converted to type A and assigned to the reference variable a of type A. Note that the type of the variable a is not type B but type A, although a is instantiated by the expression *new B()*. Therefore, a metaprocedure for this case can be implemented with the return type of A, the declared type of the variable a. The implementation of the metaprocedure for this example is shown below.

```
private A pnc()
{
    switch (mutantID)
    {
        case 1 :  return (new B());      // mutated statement 1
        case 2 :  return (new C());      // mutated statement 2
        default : return (new A());      // original code
    }
}
```

III. Instantiation Overhead

Executing a metamutated statement or function can incur overhead, because the statement or function must perform some logic to decide which version of the statement or function is to be executed. On the other hand, the original statement is executed most of the time **and** is much cheaper to run. The MSG method uses a strategy called *twinning* to improve performance of metamutants. Each statement appears in two forms: a metamutated form and the original form. The original form is used when no mutant is needed for that statement, saving significant execution time on every execution of a mutant.

This is implemented by assigning a unique ID to each method and to each statement of the program. Each mutant is then associated with a `MethodID` and `StatementID`. Consider the example shown below. If a method whose method ID is 3 is executed with a mutant whose method ID is 4, the original form of the 3rd method is executed. However, if the mutant's statement ID is 8, the metamutated form of that statement is executed.

```
if (MethodID == 3)
{
    if (StatementID == 8)
        {  metamutated form of the 8th statement;  }
    else
        {  original form of the 8th statement;  }

    ...  ...
}
else
{  original form of the 3rd method;  }
```

IV. Statements with Few Mutants
In traditional non-OO mutation analysis, most statements have many mutants. With the OO operators, however, many statements only have a few mutants. Using metaprocedures for statements that only have a few mutants imposes unnecessary overhead from the method call. For example, the ISK mutation operator produces only one mutant; the keyword *super* is deleted or inserted. For an ISK mutant, an *if-else* is more efficient than calling a metaprocedure.

3.2 Generating and Running Structural Mutants

Structural mutants change aspects of the program's structure such as variables and method declarations. Non-OO mutation operators have no need to modify program structure. The MSG method is not appropriate to change the program structure, because the case structure cannot change declarations of variable and methods.

MuJava uses bytecode translation to change data structures. Changing the bytecode directly means that additional compilation is not required. MuJava uses BCEL [15, 22] because it supports all structural mutation operators, in contrast to limitations imposed by other bytecode translation toolkits. The following subsections briefly describe BCEL and how it is used for mutation analysis.

3.2.1 Byte Code Engineering Library (BCEL)

The BCEL API helps developers make changes at a high level of abstraction through three components:

1. A package of classes that describe "static" constraints on class files. This reflects the class file format and is not intended for bytecode modifications. This is useful for analyzing Java classes without requiring the source files to be available. The top-level data structure is a JavaClass, which consists of fields, methods, and symbolic references to the parent class and to interfaces that the class implements.

2. A package to dynamically generate or modify bytecode. It can be used to insert analysis code, to strip unnecessary information from class files, and to implement the code generator back-end of a Java compiler. For example, the ClassGen class offers an interface to add methods, fields, and attributes to a class.

3. Various code examples and utilities such as a class file viewer, a tool to convert class files into HTML, and a converter from class files to the Jasmin assembly language [46].

MuJava uses the first and second components, but does not need the third. The first component is used to analyze the original bytecode, then extract information about the location of the bytecode to be mutated and which mutation operator to apply. The second component is used to generate mutants by modifying the bytecode.

3.2.2 Generating Structural Mutants

The overall process of generating and running structural mutants is illustrated in Figure 3. Major components from Figure 1 are elaborated inside the *structural mutants generator* and *structural mutants executor* boxes.

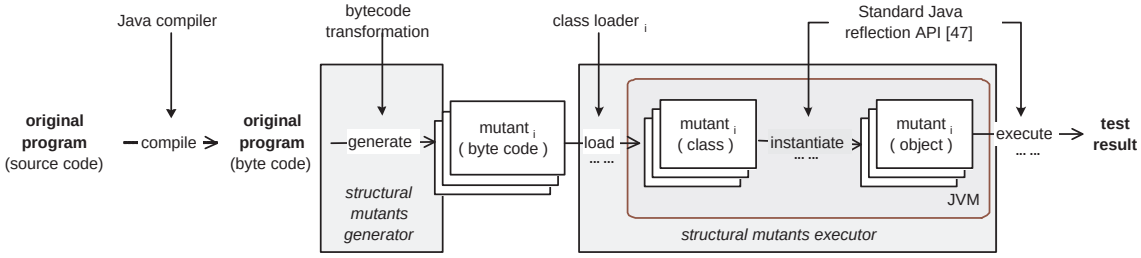

Figure 3: **Mutation Process for Structural Mutants**

To apply bytecode translation, the original source code is compiled by a Java compiler. The program structure is modified in the resulting bytecode by using the BCEL API to add and delete class members. As an example, Figure 4 shows a sequence of statements that generates IOD structural mutants. The IOD mutation operator deletes overriding methods.

```
class Parent
{
    public void temp()
    {
        System.out.println ("This is a parent");
    }
}

class Child extends Parent
{
    public void temp()
    {
        System.out.println ("This is a child");
    }
}
```

Figure 4: *Parent* **and** *Child* **Classes**

The example in Figure 4 has two classes, *Parent* and *Child*. *Child* has an overriding method *temp()*. The IOD operator generates a mutant by deleting *temp()* in *Child*. The statements to implement IOD are shown in Figure 5. It first gets the bytecode of *Child* then determines if *Child* has any overriding methods. If it does, a new copy of the bytecode for *Child* is produced without *temp()*.

Other structural mutants are produced in a similar fashion. By modifying the bytecode, structural mutant generation only requires one compilation.

```
JavaClass childClassObj = Repository.lookupClass ("Child");
Method[] methods = childClassObj.getMethods();
for (int i=0; i<methods.length; i++)
{
    if (isOverridingMethod (methods[i]))
    {
        ClassGen cgen = new ClassGen (childClassObj);
        cgen.removeMethod (methods[i]);
        File mutantF = new File (MUTANT_DIR, childClassObj.getClassName()+".class");
        cgen.getJavaClass().dump (new File ("mutant"));
    }
}
```

Figure 5: **Example Code for IOD Mutant Generation**

Mutated versions of the bytecode have the same name as the original class file, and are saved in a different directory. The JVM loads Java classes through class loaders. However, there is no support in Java to redefine a class that has already been loaded. Therefore, in order to load mutant classes that have the same name as the original class, different class loaders are assigned to each mutant. Loaded mutant classes are then instantiated and executed against the tests using Java reflection [47].

3.3 The MuJava Tool

The Java class mutation operators in Table 1 have been implemented in a tool called MuJava (**Mu**tation System for **Java**). MuJava is the result of a collaboration between two universities, Korea Advanced Institute of Science and Technology (KAIST) in South Korea and George Mason University in the USA. MuJava is available for experimental and educational use and the MuJava Web site is mirrored at both universities; `http://salmosa.kaist.ac.kr/LAB/mujava/` at KAIST and `http://www.ise.gmu.edu/~ofut/mujava/` at GMU. The Web sites have links to download the MuJava jar files, a description of the tool, and detailed instructions as to how to install and use MuJava.

MuJava has three major functions: (1) generate mutants, (2) analyze mutants, and (3) run test cases supplied by the tester. To compile mutants, MuJava uses the class `com.sun.tools.javac.Main` included in JDK. Test cases are supplied by the tester in the form of methods that contain sequences of calls to methods in the class. Each test method should have no parameters and return a string result that is used to compare outputs of mutants with outputs of the original class. Each test method should start with the string "test" and have public access. A small example test set for a Stack class is shown in Figure 9. MuJava provides a graphical user interface for each function, which consists of three tabbed panels as shown in Figures 6, 7 and 8.

Figure 6 shows the interface for generating mutants with class mutation operators as well as traditional mutation operators. Testers can select the files for which they want to create mutants and choose which mutation operators to apply. Pressing the "Generate" button prompts the tool to generate mutants. After generation, the information for each mutant is shown in the "Mutants Viewer" tabs.

Figure 7 shows the interface for analyzing mutants, which displays the portions of the original source code that are changed by the mutant. It is subdivided into a part for viewing class mutants and another for traditional mutants. The interface for the class mutants is further divided into two parts because MuJava generates mutants in two different ways (MSG and bytecode translation).

The upper part shows behavioral mutants generated with MSG. All behavioral mutants are encoded into one *metamutant*, and reading this code is very difficult. This graphical aid simplifies the viewing. By selecting a mutant listed at the left, the tester can see the mutated code on the right side highlighted in red. This helps testers both to design tests to kill mutants and to identify equivalent mutants.

The lower part of the screen in Figure 7 shows structural mutants generated with bytecode translation.

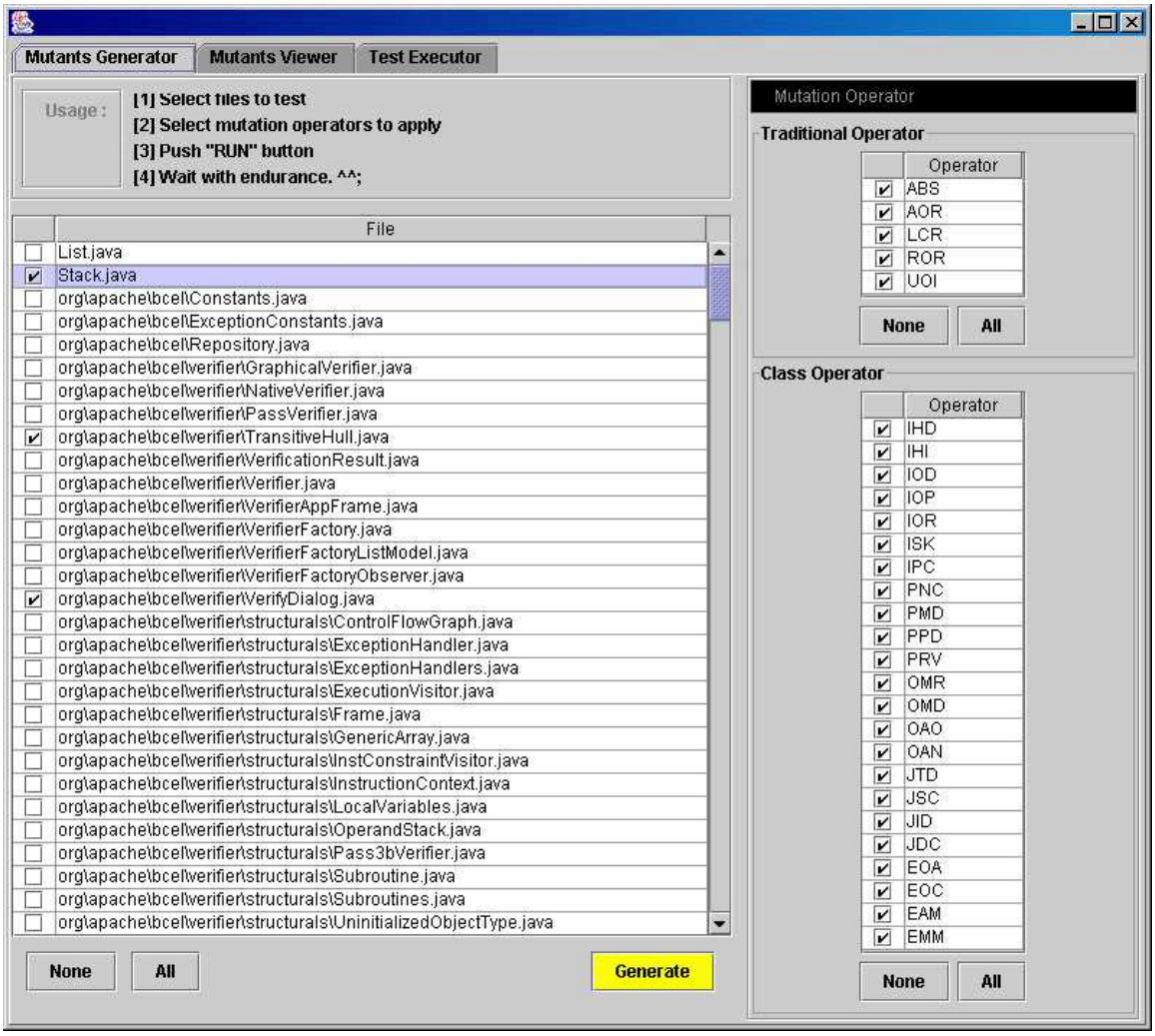

Figure 6: **GUI for Generating Mutants**

Structural mutants are generated directly from the bytecode, thus the mutated part cannot be displayed directly. Instead, brief descriptions of the mutated part are displayed by selecting a mutant name and number on the left. Future work includes developing algorithms to show source code level representations of structural mutants.

The interface in Figure 8 shows the interface for running tests. It executes mutants against the test set and shows the test result in the form of a mutation score of the test set. Pressing the "Run" button causes the mutants to be executed against the test set. The results of the mutation testing and information about the live and dead mutants are shown on the lower right hand side.

4 Experimental Performance Evaluation

A major goal of this research is to empirically determine whether a mutation tool that uses a combination of MSG and bytecode translation gives better performance than a mutation tool that uses compile-time reflection. A secondary goal is to evaluate how many mutants are created by each operator. A third was to

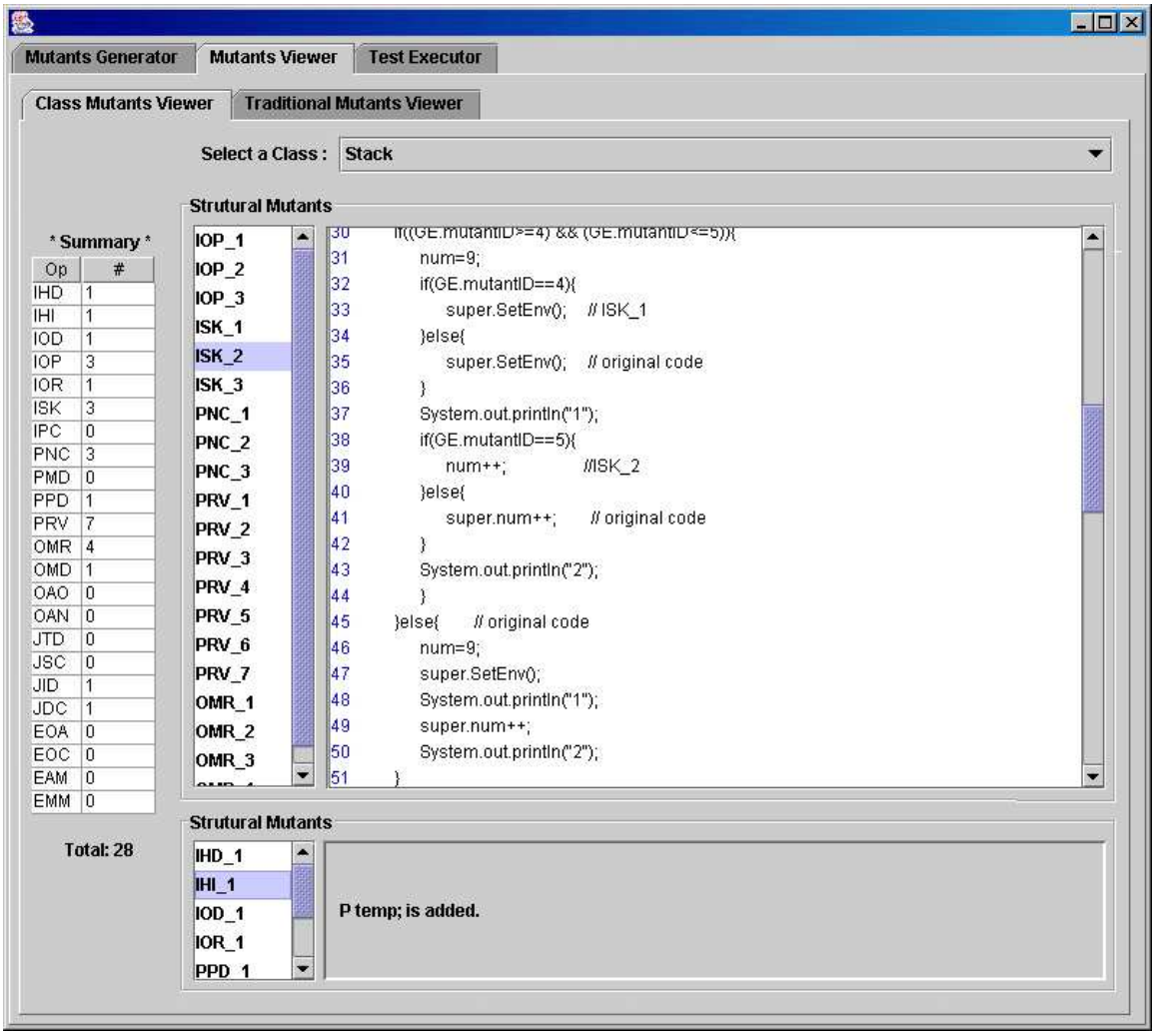

Figure 7: **GUI for Analyzing Mutants**

compare performance of the MSG method with bytecode translation. Performance is compared between the MSG/bytecode translation mutation tool MuJava and the compile-time reflection tool described previously [42]. Both tools are implemented in the same programming language and support the same language and mutation operators, which reduces threats to internal validity of the results. The two tools are only different in the design and implementation, allowing a direct performance comparison to be made.

The same programs were executed by both mutation tools and execution times for each system are compared. This experiment only considers class mutation operators. One additional threat to internal validity is that the JDK compiler is used. It has been suggested that the JDK compiler is slow, introducing a bias against the compile-time technique; however, this is also expected to be the most common tool used by testers, so the results reflect expected usage. The experimental subjects, procedure, and results are described below.

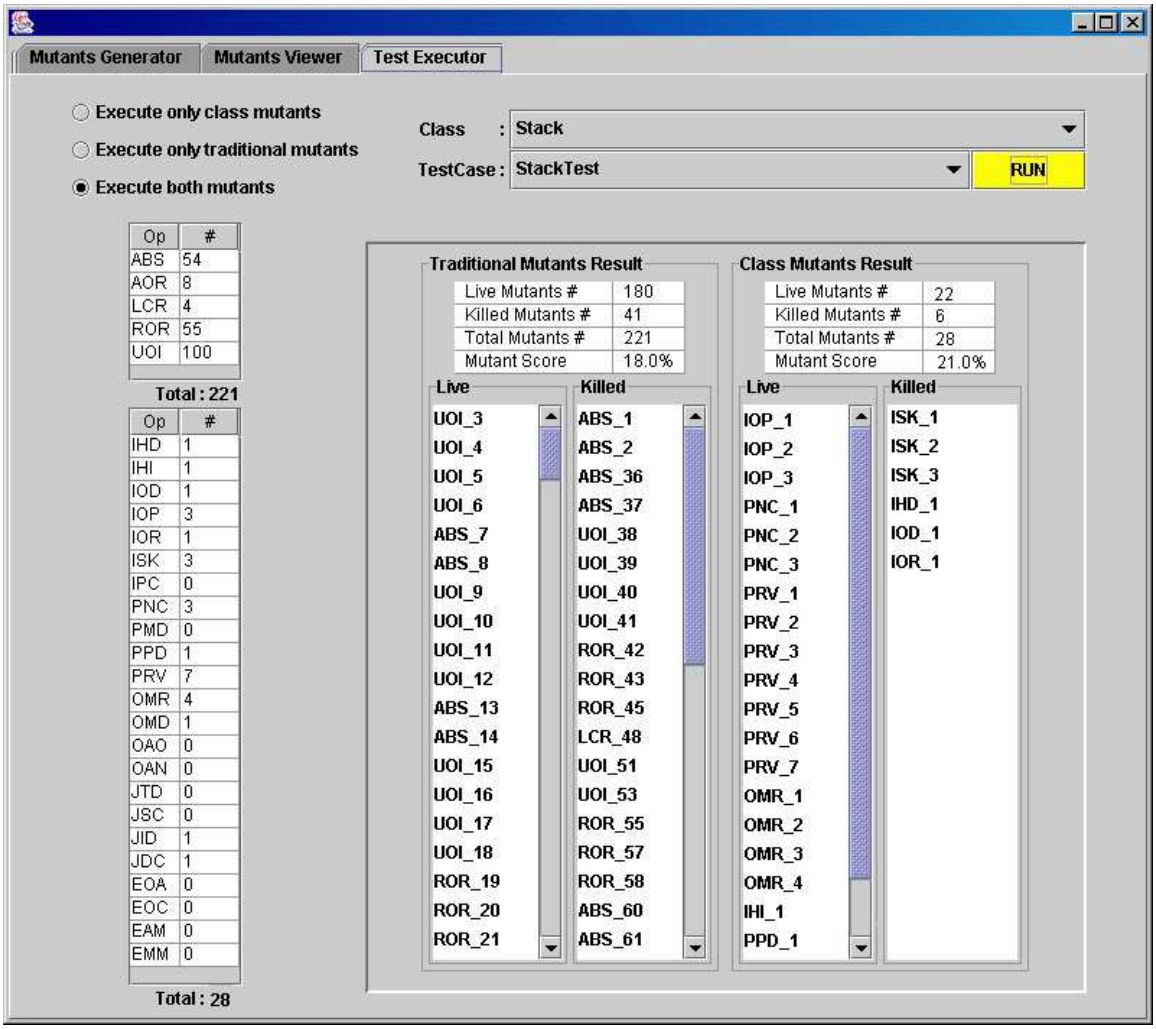

Figure 8: **GUI for Generating and Executing Test Cases**

4.1 Experimental Subjects and Numbers of Mutants

For comparison, a collection of open source classes were downloaded. The Byte Code Engineering Library (BCEL) [22] system from Apache / Jakarta is an open source project created to analyze, create, and manipulate Java bytecode files. This subject system was chosen because it is widely used, and large enough to give good results without being impractical to use. Not counting *abstract* and *interface* classes, BCEL contains 264 Java classes. Inter-class mutation operators were applied to each of the 264 classes, resulting in a total of 3,812 mutants. The mean number of mutants per class was 14.44, with a standard deviation of 47.05, and there were approximately 1.53 times more behavioral mutants than structural mutants.

The number of mutants per mutation operator is shown in Table 2. For those knowledgeable of mutation, it is worth pointing out that the total number of class mutants is relatively small. Traditional mutation operators usually result in hundreds of mutants for methods of 20 to 30 lines. For example, 951 mutants are generated for the commonly studied 30-line triangle classification program TriTyp [49]. For the traditional operators used in Mothra, the number of mutations has been quantified as $O(Vals \times Refs)$ [48], where *Vals* is the number of data objects and *Refs* is the number of data references. However, in BCEL, the average

```
public class StackTest
{
    public String test1 ()
    {
        String result;
        Stack obj = new Stack();
        obj.push (2);
        obj.push (4);
        result = “”+ obj.isFull()+obj.pop();
        return result;
    }
    public String test2 ()
    {
        String result;
        Stack obj = new Stack();
        obj.push (5);
        obj.push (3);
        result = “”+ obj.pop()+obj.pop();
        return result;
    }
}
```

Figure 9: An Example Test Set for Class Stack

number of class mutants for 264 classes is only 14.44 and the average number of lines per class is 84.7. Although the number of class mutants has not yet been quantified, it is clear from the subject programs that far fewer mutants are created with class mutation operators. The actual number depends on the use of OO language features and will be determined as this research further matures.

One point to note is that the distribution of the class mutants is not even. For example, no mutants are generated by the IHD, PMD, PPD, and ISK operators, while 1,016 mutants are generated for the EAM operator (almost one third of the total). Although the number of mutants created by each mutation operator reflects the usage of OO language feature related with the operator, the small numbers for some operators do **not** necessarily indicate the operators are useless. It is possible that they will cause tests to be created that are very effective at detecting specific kinds of faults, as has been the case with statement mutation operators. The availability of the MuJava tool will allow experimentation to be carried out to answer these kinds of questions.

4.2 Experimental Procedure

It being impractical to generate tests for all 264 BCEL classes, a smaller set was used to carry out the performance comparison. Seven classes were chosen according to two criteria. The first was to try to choose classes that are as “representative” as possible. Of course this brings up a perpetual problem with empirical software engineering research: there is no formal notion of how to choose a representative sample of programs. This study attempted to choose representative classes by choosing classes with a variety of sizes. That is, the sizes (in terms of lines, variables, methods and inheritance depth) of the classes are reasonably distributed among the 264 BCEL classes. Of course, any selection introduces a threat to external validity whose effect is difficult to analyze.

A second criterion for selection was based on the type of mutant. More than 60% of the class mutants are behavioral, and many of the classes had few structural mutants. For the performance comparison, seven BCEL classes of varying sizes were chosen, with the requirement that each had at least eight structural mutants. These seven classes and the number of class mutants for each are summarized in Table 3.

Test sets were created to kill all class mutants for each class. A test set is comprised of test cases and

Mutation operators		Total # of mutants	Mean # of mutants
Structural operators	IHD	0	0.00
	IHI	96	0.36
	IOD	496	1.88
	IOR	497	1.88
	IPC	173	0.66
	PMD	0	0.00
	PPD	0	0.00
	OMD	10	0.04
	JSC	11	0.04
	JID	115	0.44
	JDC	106	0.40
Sub-Total		**1,504**	**5.70**
Behavioral operators	IOP	44	0.17
	ISK	0	0.00
	PNC	243	0.92
	PRV	289	1.09
	OMR	59	0.22
	OAN	53	0.20
	JTD	203	0.77
	EOA	4	0.02
	EOC	5	0.02
	EAM	1,016	3.85
	EMM	392	1.48
Sub-Total		**2,308**	**8.74**
Total		**3,812**	**14.44**

Table 2: Class Mutants for 264 BCEL Classes

each test case is a sequence of method calls. The "methods" column in Table 3 gives the number of methods in each class (with the number of constructors in parentheses), and the "test size" column gives the average (mean) number of method calls per test for each class. (Only calls to methods in the class under test were counted, excluding the constructors.) Thus, each test case can cover many statements, branches, and kill many mutants. The test sets also satisfied branch coverage for all methods. The tests were created by hand. First, tests to execute every method at least once were generated. Then live mutants were examined and tests to kill them were generated. This procedure was repeated until all mutants were killed. The number of test cases needed for each class is shown under the "test cases" column in Table 3.

These seven Java classes were tested with both mutation systems. The same tests and the same set of mutants were used for both systems. The total execution times of each class for all mutants and all tests under each mutation system were calculated by the following formula:

Total Time = Mutants Generation Time + Mutants Execution Time

The *mutants generation time* is the time needed to generate all mutants. The *mutants execution time* for each class is the execution time of all tests on all the mutants plus the time for producing the mutation results (mutation score). The number of executions is calculated by multiplying the number of mutants by the number of test cases. Class c7, for example, had 110 mutants and 51 test cases, resulting in 5,610 executions.

Class	Lines	Vars	Methods (Const.)	Inher. depth	Test cases	Test size	Mutants		
							Structural	Behavioral	Total
c1	80	0	5 (3)	2	5	5.6	**8**	**3**	**11**
c2	126	0	9 (4)	2	9	1.9	**8**	**4**	**12**
c3	80	2	9 (1)	1	11	2.2	**13**	**5**	**18**
c4	319	5	14 (2)	0	17	1.4	**9**	**27**	**36**
c5	205	2	33 (1)	1	35	2.0	**64**	**1**	**65**
c6	380	13	42 (3)	1	47	2.8	**9**	**77**	**86**
c7	533	13	37 (4)	0	51	2.3	**14**	**96**	**110**

Table 3: **Summary of Seven Experimental Classes**

4.3 Experimental Results

This section presents three analyses of the difference in execution time between the two approaches. First, the combined MSG/bytecode translation method is compared with separate compilation, then the bytecode translation method (structural mutants only) is compared with separate compilation. Finally, the MSG method (behavioral mutants only) is compared with separate compilation.

4.3.1 MSG/Bytecode Translation vs. Separate Compilation

Data comparing all mutants generated and executed with the MSG/bytecode translation method and separate compilation method are shown in Table 4. For both mutation analysis techniques, the table shows the generation time, the execution time, and the total time. Speedup is calculated by dividing the time for separate compilation by the time for the MSG method.

Class	Mutants	MSG / Bytecode			Separate Compilation			Speedup		
		gen. time	exec. time	total time	gen. time	exec. time	total time	gen. time	exec. time	total time
c1	11	842	829	**1,671**	4,768	1,084	**5,852**	5.66	1.31	**3.50**
c2	12	1,016	1,022	**2,038**	5,295	1,432	**6,727**	5.21	1.40	**3.30**
c3	18	808	2,414	**3,222**	6,402	3,232	**9,634**	7.92	1.34	**2.99**
c4	36	3,672	1,311	**4,983**	19,115	3,068	**22,183**	5.21	2.34	**4.45**
c5	65	1,224	7,252	**8,476**	24,290	7,308	**31,598**	19.84	1.01	**3.73**
c6	86	3,209	1,901	**5,110**	34,283	8,659	**42,942**	10.68	4.55	**8.40**
c7	110	5,020	2,676	**7,696**	52,717	11,362	**64,079**	10.5	4.25	**8.33**
total	338	15,791	17,405	**33,196**	146,869	36,145	**183,014**	9.30	2.08	**5.51**

Table 4: **Execution Time Comparison for All Mutants**
All times in milliseconds. Gen. time = mutants generation time, exec. time = mutants execution time.

The MSG/bytecode translation method is more efficient for both mutant generation and mutant execution. The total time to generate and execute mutants with the MSG/bytecode translation method was 5.51 times faster on the average than with separate compilation. The mutant generation time and the mutant execution time were 9.30 times and 2.08 times faster on the average. In our opinion, these across the board speedups make a pretty convincing case for using this method to implement mutation for Java programs.

4.3.2 MSG vs. Separate Compilation

The times of generating behavioral mutants and executing them and the total time with the MSG method and separate compilation are shown in Table 5. Speedup is again calculated by dividing the total time for separate compilation by the total time for the MSG method.

Class	Behavioral mutants	MSG method			Separate Compilation			Speedup		
		gen. time	exec. time	total time	gen. time	exec. time	total time	gen. time	exec. time	total time
c1	3	774	41	**815**	1,440	216	**1,656**	1.86	5.27	**2.03**
c2	4	944	68	**1,011**	1,942	498	**2,440**	2.06	7.37	**2.41**
c3	5	663	80	**743**	1,911	598	**2,509**	2.88	7.51	**3.38**
c4	27	3,600	344	**3,943**	14,505	2,301	**16,806**	4.03	6.69	**4.26**
c5	1	701	56	**757**	702	92	**794**	1.00	1.64	**1.05**
c6	77	3,116	995	**4,111**	30,766	7,753	**38,519**	9.87	7.99	**9.37**
c7	96	4,797	1,243	**6,040**	45,710	9,827	**55,537**	9.53	7.79	**9.37**
total	213	14,595	2,286	**17,421**	96,977	21,284	**118,261**	6.64	7.53	**6.79**

Table 5: **Execution Time Comparison for Behavioral Mutants**
All times in milliseconds. Gen. time = mutants generation time, exec. time = mutants execution time

The MSG method was found to be more efficient for both mutant generation and test execution on the average. The average total time is 6.79 times faster than separate compilation. Both the generation time and execution time were faster, and faster by similar amounts (6.64 and 7.53 on the average). Generally speaking, the difference was greater with more mutants.

The speedup in execution time may be more surprising than the speedup in generation time, because of the overhead incurred with the MSG method. However, there is also an overhead with the separate compilation method. Specifically, each mutant requires a different class file to be loaded. Java classes are loaded through "class loaders." However, the JVM does not support redefinition of a class that is already loaded. Thus, to load a new mutated class, a new class loader is needed. Also, to avoid side effects caused by using classes that are related to the mutated class, all related classes are also reloaded through the new class loader. The MSG method only needs one class loader, because all mutants are integrated into one class file. This reduces the overhead for loading mutants with MSG.

4.3.3 Bytecode Translation vs. Separate Compilation

The times of generating structural mutants and executing them and the total time with the bytecode translation method and separate compilation are shown in Table 6.

As it turned out, the average total time of the bytecode translation method was 4.26 times faster than separate compilation. However, the speedup in generation time with bytecode translation was noticeably high, 44 times on the average, whereas the speed of execution was the same for both techniques.

The fact that there was a greater difference in the generation time should not be surprising. Generating mutants with bytecode translation involves working with efficient structures (the bytecode objects), and is done mostly in memory. Generating mutants with compile-time reflection involves parsing program source, which is more time consuming and requires more disk accesses. As an example, to find the type of a variable at the source code level, the program must make a number of inferences from the program context, but this information is directly available in the bytecode.

Class	Structural mutants	Bytecode Translation			Separate Compilation			Speedup		
		gen. time	exec. time	total time	gen. time	exec. time	total time	gen. time	exec. time	total time
c1	8	68	788	**858**	3,520	788	**4,308**	51.76	1.00	**5.03**
c2	8	72	955	**1,027**	3,619	955	**4,573**	50.26	1.00	**4.45**
c3	13	145	2,334	**2,479**	4,675	2,334	**7,009**	50.26	1.00	**2.83**
c4	9	72	967	**1,039**	5,284	967	**6,251**	32.24	1.00	**6.02**
c5	64	523	7,196	**7,719**	23,922	7,196	**31,117**	73.39	1.00	**4.03**
c6	9	93	906	**999**	4,197	906	**5,103**	45.74	1.00	**5.11**
c7	14	223	1,433	**1,9656**	7,409	1,433	**8,842**	33.23	1.00	**5.34**
total	125	1,196	15,775	**15,775**	52,624	14,579	**67,203**	44.00	1.00	**4.26**

Table 6: **Execution Time Comparison for Structural Mutants**
All times in milliseconds. Gen. time = mutants generation time, exec. time = mutants execution time.

4.3.4 Discussion

Table 5 showed that the MSG method achieved an average speedup of 6.79 times in generating and executing behavioral mutants over separate compilation and Table 6 showed that the bytecode translation method achieved an average speedup of 4.26 times in generating and executing structural mutants over separate compilation. While both the MSG method and the bytecode translation were found to be faster than separate compilation, it is interesting to note that they were faster for different reasons. The MSG method improves on both mutant generation and execution. The mutants generation time is reduced by conducting only one compilation. The mutants execution time is reduced by loading only one class, metaclass, whereas bytecode translation only improves on mutant generation but by a much bigger factor. This factor can be obtained because lots of information which must be inferred when source code is used is directly available in the bytecode.

In fact, the bytecode translation method can be used for both structural mutants and behavioral mutants in contrast to the MSG method, which can be applied only to behavioral mutants. However, we used the MSG method in our approach because, in one of our experiments conducted with some behavioral mutants, the bytecode translation method achieved a speedup comparable to the case with structural mutants. Although a direct comparison is not possible with the small experiment, the MSG method appears to be more efficient than the bytecode translation method. However, it is not correct to interpret that the MSG method should be used exclusively over bytecode translation because the MSG method cannot be used for structural mutants.

Another interesting point to note, particularly to researchers familiar with traditional mutation, is that execution times are reasonably comparable with generation times. With traditional statement-level mutation, many more tests are usually required and the generation time winds up being insignificant when compared with the execution time. This is not the case with class level mutation, primarily because there are few mutants, and the nature of the tests (sequences of method calls) means that fewer tests are needed.

4.3.5 Limitations and Threats to Validity

A strength of this comparison is that it was possible to use two mutation tools that were written in the same language and implemented the same collection of mutation operators. This is a rare luxury in empirical software engineering and eliminates internal validity threats due to environmental factors.

Some concerns were mentioned previously. The tools used the JDK compiler, which is considered to be fairly slow and thus introduces a bias against the separate compilation approach. The fact that the JDK compiler is used so widely, however, means that it is likely to be the typical way that mutation tools would be used.

As is usual in software engineering experiments, there is also the question of how representative the subjects are. This study tried hard to reduce the threat to external validity by this factor. It is encouraging, although not definitive, that the results are reasonably consistent across the subjects.

5 Related Work in Performance of Mutation

There have been a number of attempts to overcome the performance problem of mutation testing. As described in Section 1.1, they usually follow one of three strategies: *do fewer*, *do smarter*, or *do faster*.

The *do fewer* approaches try to run fewer mutant programs without incurring unacceptable information loss. Mutant sampling [1, 5, 62] uses a random sample of the mutants and is the simplest *do fewer* approach. Although this approach lowers execution cost, it also weakens the mutation adequacy criterion. Wong and Mathur suggested the idea of *selective mutation* to be applying mutation only to the most critical mutation operators being used [63, 64]. This idea was later developed by Offutt et al. [48, 52] who identified a set of selective operators for Fortran-77 with Mothra. Results showed that selective mutation provide almost the same test coverage as non-selective mutation. It might be worthwhile to identify selective OO mutation operators as well.

The use of non-standard computer architecture has been explored as a *do smarter* approach. This approach distributes the computational expense over several machines. Work has been done to adapt mutation analysis system to vector processors [45], SIMD machines [38], Hypercube (MIMD) machines [13, 50], and Network (MIMD) computers [65]. *Weak mutation* [30] is another *do smarter* approach. It is an approximation technique that compares the internal states of the mutant and original program immediately after execution of the mutated portion of the program. Experimentation has shown that weak mutation can generate tests that are almost as effective as tests generated with strong mutation, and that at least 50% and usually more of the execution time is saved [49].

The *do faster* approaches try to generate and run mutants as quickly as possible. A particular goal of the *do faster* approach is to avoid the interpretative execution, the primary reason that older mutation analysis systems were slow. In the *separate compilation* approach, each mutant is individually created, compiled, linked and run. However, unless mutant run time greatly exceeds individual compilation/link times, a system based on such a strategy will experience a *compilation bottleneck* [13]. *Compiler-integrated* program mutation [18] seeks to avoid excessive compilation overhead and yet retain the benefit of compiled speed execution by directly modifying linked object code. However, crafting the special compiler needed turns out to be very expensive and difficult. Another *do faster* approach is the Mutant Schema Generation (MSG) [60] method that was used in this research.

6 Conclusions and Future Work

This paper presented three results. First was an approach and collection of mutation operators for intra-class testing of object-oriented software based on mutation. These ideas initially appeared in a conference paper [43]. Second was an application of an existing mutation *do faster* approach, the mutant schemata generation (MSG) method, to the testing of object-oriented software. MSG was used previously for C with statement-level operators and had to be adapted to be used with class-level operators. Third was a tool for applying mutation at the object-oriented level and results from using that tool. These results indicate that a combination of MSG and bytecode translation can reduce the execution time over a separately compiled approach.

Traditional mutation testing only changes the behavior of a program. To apply mutation testing to OO software, it is necessary to generate mutants that change the structure of the program. Therefore, class mutants were classified into two types, behavioral mutants and structural mutants, and a *do faster* approach was devised for each type. An existing method, MSG, was adapted for behavioral mutants. Bytecode translation is used for structural mutants.

A key advantage of this solution is that it requires only two compilations: compilation of the original source code and compilation of the MSG metamutant. This greatly reduces the time needed for mutant generation.

The mutation system MuJava was built and used to obtain empirical results. These results indicate that the MSG/bytecode translation method is more efficient than separate compilation, being about five times faster. When considering MSG and bytecode translation separately, MSG was faster both for generating and executing behavioral mutants, whereas bytecode translation was much faster for generating structural mutants but not for execution. Although a direct comparison between the MSG and the bytecode translation methods is not possible because the current tool uses the two methods to create different type of mutants, they could be compared with just the behavioral mutants by creating them with the bytecode translation technique. Preliminary data indicated that the MSG method may be more efficient, but the MSG is inherently limited by the fact that it cannot change the program structure, so cannot be used for structural mutants.

One difficulty with the MSG method is handling of persistent parts of the state. All mutants generated by the MSG method run within the same process. This makes it difficult to ensure that each execution starts with all objects in the same state, because some objects may have parts of their state (that is, variables) that have values left over from previous mutants. This is not an issue with bytecode translation mutation systems.

Another advantage of the MSG/bytecode translation is portability. Because the bytecode translation system that MuJava uses works with any standard Java compiler and JVM, it can easily be moved between machines and compilers.

This is an ongoing research project. Although the MSG/bytecode translation approach is significantly faster than separate compilation, its speed can still be improved. In particular, the bytecode translation was faster when generating mutants, but exactly the same as compile-time mutation when executing mutants. A major portion of the execution time with bytecode translation is from the loading of different class loaders. This could be solved by using a run-time reflection system. At this time, however, the only run-time reflection systems that are available use non-standard JVMs, which are not practical for this project.

The current version of MuJava has some usability issues. MuJava currently does not display mutants in a very convenient way. The problem of showing mutants in the context of the complete class is harder than for traditional mutation systems, but it should be possible to show mutants as being embedded in the original source code.

Finally, now that MuJava is complete, several experiments are planned. An obvious experiment is to carry out an experimental evaluation of the efficacy of using mutation to test classes, using actual faults. A refinement of this study would be to determine what kinds of faults can be found by tests that satisfy OO mutation, and whether tests based on OO mutants find different faults than tests based on statement-level mutants. The OO mutants were derived from definitions of faults for subtype inheritance and polymorphism [53], so it is reasonable to expect tests from these mutants to find those kind of faults. An interesting comparison could be made with the state-modifying mutation operators of Alexander et al. [3]. It is also possible that the mutant operators could be reduced by using a selective approach. As a first step toward selective class-level mutation, the number of mutants that are generated must be quantified.

References

[1] A. T. Acree. *On Mutation.* PhD thesis, Georgia Institute of Technology, Atlanta, GA, 1980.

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

[3] Roger T. Alexander, James M. Bieman, Sudipto Chosh, and Bixia Ji. Mutation of Java objects. In *Proceedings of 13th International Symposium on Software Reliability Engineering*, pages 341–351, Annapolis MD, November 2002. IEEE Computer Society Press.

[4] Roger T. Alexander and A. Jefferson Offutt. Criteria for testing polymorphic relationships. In *Proceedings of the 11th International Symposium on Software Reliability Engineering*, pages 15–23, San Jose CA, October 2000. IEEE Computer Society Press.

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

[6] Michelle Cartwright and Martin Shepperd. An empirical investigation of an object-oriented software system. *IEEE Transactions on Software Engineering*, 26(8):786–796, August 2000.

[7] T. E. Cheatham and L. Mellinger. Testing object-oriented software systems. In *1990 ACM Eighteenth Annual Computer Science Conference*, pages 161–165, February 1990.

[8] H. Y. Chen, T. H. Tse, F. T. Chan, and T. Y. Chen. In black and white: An integrated approach to class-level testing. *ACM Transactions on Software Engineering Methodology*, 7(3):250–295, 1998.

[9] Mei-Hwa Chen and Ming-Hung Kao. Testing object-oriented programs - an integrated approach. In *Proceedings of the 10th International Symposium on Software Reliability Engineering*, pages 73–83, Boca Raton, FL, November 1999. IEEE Computer Society Press.

[10] 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)*, pages 267–270, Macau SAR, China, December 2001.

[11] Philippe Chevalley and Pascale Thévenod-Fosse. A mutation analysis tool for Java programs. *Journal on Software Tools for Technology Transfer (STTT)*, pages 1–14, December 2002.

[12] Shigeru Chiba. Load-time structural reflection in Java. *LNCS*, 1850:313–336, 2000.

[13] B. Choi and A. P. Mathur. High-performance mutation testing. *Journal of Systems and Software*, 20:135–152, Febuary 1993.

[14] Geoff A. Cohen, Jeffrey S. Chase, and David L. Kaminsky. Automatic program transformation with JOIE. In *1998 USENIX Annual Technical Symposium*, pages 167–178, New Orleans, Louisiana, USA, 1998.

[15] M. Dahm. Byte code engineering with the JavaClass API. *Technical Report B-17-98, Institut fur Informatik, Freie Universitat Barlin*, January 1999.

[16] 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.

[17] Marcio Delamaro, Jose Maldonado, and Auri Vincenzi. Proteum/IM 2.0: An integrated mutation testing environment. In *Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 91–101, San Jose, CA, October 2000.

[18] R. A. DeMillo, E. W. Krauser, and A. P. Mathur. Compiler-integrated program mutation. In *Proceedings of the Fifteenth Annual Computer Soctware and Applications Conference*, pages 351–356, September 1991.

[19] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[20] S. P. Fiedler. Object-oriented unit testing. *Hewlett-Packard Journal*, 40(2):69–74, April 1989.

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

[22] Apache Software Foundation. BCEL : Byte Code Engineering Library WWW page. Part of the Apache/Jakarta project, 2002-2003. http://jakarta.apache.org/bcel/ (accessed May 2004).

[23] L. Gallagher and A. J. Offutt. Integration Testing of Object-oriented Components Using FSMS: Theory and Experimental Details. Technical report ISE-TR-04-04, Department of Information and Software Engineering, George Mason University, Fairfax, VA, July 2004. http://www.ise.gmu.edu/techrep/.

[24] J. Z. Gao, D. Kung, P. Hsia, Y. Toyoshima, and C. Chen. Object state testing for object-oriented programs. In *19th Computer Software and Applications Conference (COMPSAC 95)*, pages 232–238, Dallas, TX, August 1995.

[25] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.

[26] M. J. Harrold and J. D. McGregor. Incremental testing of object-oriented class structures. In *14th International Conference on Software Engineering*, pages 68–80, Melbourne, Australia, May 1992. IEEE Computer Society Press.

[27] M. J. Harrold and Gregg Rothermel. Performing data flow testing on classes. In *Symposium on Foundations of Software Engineering*, pages 154–163, New Orleans, LA, December 1994. ACM SIGSOFT.

[28] H. S. Hong, Y. R. Kwon, and S. D. Cha. Testing of object-oriented programs based on finite state machines. In *Proceedings of the Asia-Pacific Software Engineering Conference*, pages 234–241, 1995.

[29] James Hook and Tim Sheard. A semantics of compile-time reflection. Technical Report CSE 93-019, Oregon Graduate Institute, 1993.

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

[31] X. Jia. Model-based formal specification directed testing of abstract data types. In *Proceedings of the Computer Software and Applications Conference*, pages 360–366, 1993.

[32] Ralph Keller and Urs Hölzle. Binary component adaptation. In *12th European Conference on Object-Oriented Programming*, pages 307–329. Springer LNCS 1445, 1998.

[33] G. Kiczales, J. de Rivieres, and D. G. Bobrow. *The Art of the Metaobject protocol*. MIT Press, 1991.

[34] S. Kim, J. Clark, and J. McDermid. Assessing test set adequacy for object oriented programs using class mutation. In *Proceedings of Symposium on Software Technology (SoST'99)*, pages 72–83, Sept. 1999.

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

[36] J. Kleninoder and M. Golm. MetaJava: An efficient run-time meta architecture for Java. In *Proceedings of the International Workshop on Object-Orientation in Operating Systems*, pages 54–61, 1996.

[37] Günter Kniesel and Pascal Costanza. Jmangler – A Framework for Load-Time Transformation of Java Class Files. In *IEEE Workshop on source Code Analysis and Manipulation*, pages 100–110, November 2001.

[38] E. W. Krauser, A. P. Mathur, and V. J. Rego. High performance software testing on SIMD machine. *IEEE Transactions on Software Engineering*, 17(5):403–423, May 1991.

[39] D. Kung, J. Gao, Pei Hsia, Y. Toyoshima, and C. Chen. A test strategy for object-oriented programs. In *19th Computer Software and Applications Conference (COMPSAC 95)*, pages 239 –244, Dallas, TX, August 1995. IEEE Computer Society Press.

[40] D. Kung, N. Suchak, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. On object state testing. In *Eighteenth Annual International Computer Software & Applications Conference*, pages 222–227, Los Alamitos, CA, 1993. IEEE Computer Society Press.

[41] 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.

[42] Y. S. Ma and Y. R. Kwon. A study on method and tool of mutation analysis for object-oriented program. *Software Engieering Review*, 15(3):41–52, September 2002.

[43] Y. S. Ma, Y. R. Kwon, and J. Offutt. Inter-class mutation operators for Java. In IEEE Computer Society Press, editor, *13th International Symposium on Software Reliability Engineering*, pages 352–363, Annapolis MD, November 2002.

[44] Pettie Maes. Concepts and experiments in computational reflection. In *Preceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, volume 22 (12), pages 147–155, Orlando, FA, October 1987.

[45] A. P. Mathur and E. W. Krauser. Mutant unification for improved vectorization. Technical Report SERC-TR-14-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, Apr. 25 1988.

[46] J. Meyer and T. Downing. *Java Virtual Machine*. O'Reilly, 1997.

[47] Sun Microsystems. Java reflection WWW page. Part of the Sun J2SE project, 2002. http://java.sun.com/j2se/1.4/docs/guide/reflection/ (accessed May 2004).

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

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

[50] A. J. Offutt, R. P. Pargas, S. V. Fichter, and P. K. Khambekar. Mutation testing of software using a MIMD computer. In *1992 International Conference on Parallel Processing*, pages II–257–266, Chicago, Illinois, August 1992.

[51] 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/.

[52] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *Proceedings of the Fifteenth International Conference on Software Engineering*, pages 100–107, May 1993.

[53] 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.

[54] A. Orso and M. Pezze. Integration testing of procedural object oriented programs with polymorphism. In *Proceedings of the Sixteenth International Conference on Testing Computer Software*, pages 103–114, Washington DC, June 1999. ACM SIGSOFT.

[55] D. E. Perry and G. E. Kaiser. Adequate testing and object-oriented programming. *Journal of OOP*, 2:13–19, Jan/Feb 1990.

[56] M. Tatsubori. OpenJava WWW page. Tokyo Institute of Technology, Chiba Shigeru Group, 2002. http://www.csg.is.titech.ac.jp/∼mich/openjava/ (accessed May 2004).

[57] 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.

[58] C. D. Turner and D. J. Robson. The state-based testing of object-oriented programs. In *Proceedings of the 1993 IEEE Conference on Software Maintenance (CSM-93)*, pages 302–310, Montreal, Quebec, Canada, September 1993.

[59] R. Untch. *Schema-based Mutation Analysis: A New Test Data Adequacy Assessment Method.* PhD thesis, Clemson University, Clemson SC, 1995. Clemson Department of Computer Science Technical report 95-115.

[60] R. Untch, A. J. Offutt, and M. J. Harrold. Mutation analysis using program schemata. In *Proceedings of the 1993 International Symposium on Software Testing, and Analysis*, pages 139–148, Cambridge MA, June 1993.

[61] Ian Welch and Robert J. Stroud. Kava - Using byte code rewriting to add behavioral reflection to Java. Technical report 704, University of Newcastle upon Tyne, 2000.

[62] W. E. Wong. *On Mutation and Data Flow.* PhD thesis, Purdue University, West Lafayette, December 1993.

[63] W. E. Wong, M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Constrained mutation in C programs. In *Proceedings of the 8th Brazilian Symposium on Software Engineering*, pages 439–452, October 1994.

[64] W. E. Wong and A. P. Mathur. Reducing the cost of mutation testing: An empirical study. *The Journal of Systems and Software*, 31(3):185–196, Dec. 1995.

[65] C. N. Zapf. MedusaMothra - A distributed interpreter for the Mothra mutation testing system. M.S. Thesis, Clemson University, Clemson, SC, Aug. 1993.

A Number of Class Mutants for all 264 Classes

This appendix gives data on all 264 classes in BCEL. For each class, the number of lines, variables, methods and inheritance depth is given, the number of mutants for each type is given, and the total number of mutants for that class. The last page of the appendix gives a total for all 264 classes. The seven classes that were used in the experiment are highlighted in bold (class numbers 22, 30, 53, 58, 61, 216 and 252).

classes	Lines	Veriables	Method (Constructors)	Inheritance depth	Mutants																							
					I HD	I H I	IOD	IOP	IOR	ISK	IPC	PNC	PMD	PPD	PRV	OMR	OMD	OAN	JTD	JSC	JID	JDC	EOA	EOC	EAM	EMM	Total
c1	21	0	0 (2)	3	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**2**
c2	174	15	9 (3)	0	0	0	0	0	0	0	0	0	0	0	6	0	0	0	5	1	0	0	0	0	0	0	**12**
c3	217	8	19 (3)	1	0	3	2	2	2	0	0	0	0	0	26	1	0	2	8	0	0	0	0	0	20	0	**66**
c4	126	4	14 (3)	0	0	0	1	0	1	0	0	0	0	0	24	3	0	2	8	0	0	0	0	0	12	0	**51**
c5	72	1	7 (3)	1	0	1	1	0	1	0	0	0	0	0	0	0	0	1	2	0	0	0	0	0	0	0	**6**
c6	66	1	6 (3)	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	**5**
c7	36	0	1 (3)	2	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	**4**
c8	66	1	6 (3)	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	**5**
c9	66	1	6 (3)	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	**5**
c10	36	0	1 (3)	2	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	**4**
c11	66	1	6 (3)	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	**5**
c12	36	0	1 (3)	2	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	**4**
c13	85	2	9 (3)	1	0	1	1	0	1	0	0	0	0	0	4	0	0	0	4	0	0	0	0	0	4	0	**15**
c14	229	2	14 (2)	0	0	0	1	0	1	0	0	20	0	0	0	1	0	5	1	0	0	0	1	0	0	0	**30**
c15	72	1	7 (3)	1	0	1	1	0	1	0	0	0	0	0	0	0	0	1	2	0	0	0	0	0	0	0	**6**
c16	65	1	5 (3)	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	**5**
c17	93	1	6 (3)	1	0	3	2	1	2	0	0	0	0	0	2	0	0	1	2	0	0	0	0	0	6	0	**19**
c18	78	1	6 (3)	1	0	3	2	1	2	0	0	0	0	0	0	0	0	2	2	0	0	0	0	0	2	0	**14**
c19	326	3	36 (1)	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	2	0	1	0	0	0	0	0	**4**
c20	147	0	32 (1)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**1**
c21	103	2	8 (3)	1	0	3	2	2	2	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	4	0	**14**
c22	**80**	**0**	**5 (3)**	**2**	**0**	**5**	**0**	**0**	**0**	**0**	**3**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**3**	**0**	**11**
c23	134	4	13 (3)	0	0	0	1	0	1	0	0	0	0	0	24	1	0	0	8	0	0	0	0	0	12	0	**47**
c24	87	2	6 (3)	1	0	3	2	2	2	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	2	0	**12**
c25	578	22	51 (2)	1	0	1	1	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	**4**
c26	82	2	8 (3)	0	0	0	1	0	1	0	0	0	0	0	4	0	0	0	4	0	0	0	0	0	2	0	**12**
c27	134	2	8 (3)	1	0	3	2	2	2	0	0	0	0	0	12	0	0	0	1	0	0	0	0	0	4	0	**26**
c28	153	6	18 (3)	0	0	0	1	0	1	0	0	0	0	0	40	0	0	2	12	0	0	0	0	0	22	0	**78**
c29	105	2	8 (3)	1	0	3	2	2	2	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	4	0	**14**
c30	**126**	**0**	**9 (4)**	**2**	**0**	**5**	**0**	**0**	**0**	**0**	**3**	**0**	**0**	**0**	**0**	**0**	**0**	**2**	**0**	**0**	**0**	**0**	**0**	**0**	**2**	**0**	**12**
c31	92	2	10 (3)	1	0	3	2	2	2	0	0	0	0	0	8	0	0	2	4	0	0	0	0	0	14	0	**37**
c32	176	1	14 (3)	1	0	0	2	1	2	0	0	30	0	0	2	0	0	1	2	0	0	0	0	0	6	0	**46**
c33	72	1	7 (3)	1	0	3	2	1	2	0	0	0	0	0	2	0	0	1	2	0	0	0	0	0	6	0	**19**
c34	91	2	7 (2)	1	0	3	2	2	2	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	**10**
c35	167	6	16 (2)	0	0	0	1	0	1	0	0	0	0	0	16	0	0	0	7	0	0	0	0	0	0	0	**25**
c36	112	3	11 (2)	0	0	0	1	0	1	0	0	10	0	0	0	0	0	0	1	0	1	0	0	0	0	0	**14**
c37	82	1	6 (3)	1	0	3	2	1	2	0	0	0	0	0	0	0	0	2	2	0	0	0	0	0	2	0	**14**
c38	111	3	8 (3)	1	0	3	2	1	2	0	0	0	0	0	0	0	0	2	2	0	1	0	0	0	2	0	**15**
c39	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**

Table 7: **Number of Mutants for Classes #1 through #40**

classes	Lines	Veriables	Method (Constructors)	Inheritance depth	Mutants																						
					I HD	I H I	IOD	IOP	IOR	ISK	IPC	PNC	PMD	PPD	PRV	OMR	OMD	OAN	JTD	JSC	JID	JDC	EOA	EOC	EAM	EMM	Total
c41	29	0	2 (1)	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c42	27	0	1 (2)	3	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**
c43	51	0	3 (2)	2	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**2**
c44	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c45	28	0	2 (1)	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c46	90	2	5 (3)	2	0	0	1	0	1	0	1	0	0	0	2	0	0	0	3	0	0	0	0	0	0	0	**8**
c47	27	0	1 (2)	3	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**
c48	28	0	2 (1)	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c49	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c50	63	0	2 (1)	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**
c51	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c52	65	1	6 (2)	1	0	1	3	1	3	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	**10**
c53	**80**	**2**	**9 (1)**	**1**	**0**	**2**	**5**	**2**	**5**	**0**	**0**	**0**	**0**	**0**	**3**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**18**
c54	21	0	1 (1)	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c55	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c56	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c57	51	0	3 (2)	2	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**2**
c58	**380**	**13**	**42 (3)**	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**22**	**0**	**0**	**0**	**9**	**0**	**8**	**0**	**0**	**0**	**43**	**3**	**86**
c59	21	0	0 (2)	3	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**2**
c60	119	4	13 (1)	0	0	0	2	0	2	0	0	0	0	0	6	0	0	0	5	0	0	0	0	0	0	12	**27**
c61	**533**	**13**	**37 (4)**	**0**	**0**	**0**	**1**	**0**	**1**	**0**	**0**	**20**	**0**	**0**	**35**	**0**	**0**	**0**	**0**	**4**	**8**	**0**	**3**	**0**	**38**	**0**	**110**
c62	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c63	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c64	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c65	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c66	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c67	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c68	29	0	2 (1)	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c69	29	0	2 (1)	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c70	51	1	3 (2)	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**2**
c71	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c72	27	0	1 (2)	3	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**
c73	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c74	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c75	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c76	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c77	27	0	1 (2)	3	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**
c78	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c79	24	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c80	24	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**

Table 8: **Number of Mutants for Classes #41 through #80**

classes	Lines	Veriables	Method (Constructors)	Inheritance depth	Mutants																					
					I HD	I H I	IOD	IOP	IOR	ISK	IPC	PNC	PMD	PPD	PRV	OMR	OMD	OAN	JTD	JSC	JID	JDC	EOA	EOC	EAM	EMM	Total
c81	22	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c82	22	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c83	22	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c84	22	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c85	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c86	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c87	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c88	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c89	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c90	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c91	29	0	2 (1)	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c92	29	0	2 (1)	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c93	55	1	3 (2)	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**2**
c94	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c95	242	2	21 (2)	2	0	3	1	0	1	0	0	10	0	0	1	0	5	0	0	0	1	0	0	0	13	7	**42**
c96	27	0	1 (2)	3	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**
c97	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c98	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c99	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c100	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c101	27	0	1 (2)	3	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**
c102	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c103	51	0	3 (2)	4	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**
c104	50	0	3 (2)	4	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**
c105	48	0	3 (2)	3	0	0	2	0	2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**5**
c106	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c107	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c108	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c109	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c110	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c111	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c112	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c113	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c114	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c115	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c116	47	1	3 (2)	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**2**
c117	31	0	2 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c118	33	0	2 (2)	3	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**
c119	33	0	2 (2)	3	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**
c120	33	0	2 (2)	3	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**

Table 9: **Number of Mutants for Classes #81 through #120**

classes	Lines	Veriables	Method (Constructors)	Inheritance depth	Mutants																						
					I HD	I H I	IOD	IOP	IOR	ISK	IPC	PNC	PMD	PPD	PRV	OMR	OMD	OAN	JTD	JSC	JID	JDC	EOA	EOC	EAM	EMM	Total
c121	33	0	2 (2)	3	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
c122	33	0	2 (2)	3	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
c123	33	0	2 (2)	3	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
c124	33	0	2 (2)	3	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
c125	33	0	2 (2)	3	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
c126	33	0	2 (2)	3	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
c127	33	0	2 (2)	3	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
c128	33	0	2 (2)	3	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
c129	33	0	2 (2)	3	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
c130	33	0	2 (2)	3	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
c131	33	0	2 (2)	3	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
c132	33	0	2 (2)	3	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
c133	33	0	2 (2)	3	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
c134	108	2	9 (2)	2	0	1	5	0	5	0	1	0	0	0	1	0	0	0	5	0	0	0	0	0	0	2	20
c135	27	0	1 (2)	3	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
c136	21	0	1 (1)	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	2
c137	21	0	1 (1)	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	2
c138	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	2
c139	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	2
c140	45	0	3 (2)	2	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2
c141	716	3	38 (3)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
c142	195	7	23 (1)	0	0	0	1	0	1	0	0	0	0	0	4	1	0	0	1	0	2	0	0	0	0	0	10
c143	831	6	64 (5)	0	0	0	1	0	1	0	0	0	0	0	0	32	4	0	0	0	0	0	0	0	0	0	38
c144	91	1	7 (2)	4	0	0	4	2	4	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	13
c145	48	0	2 (2)	4	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
c146	46	0	2 (2)	4	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
c147	48	0	2 (2)	4	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
c148	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	2
c149	31	0	2 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	2
c150	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	2
c151	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	2
c152	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	2
c153	27	0	1 (2)	3	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
c154	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	2
c155	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	2
c156	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	2
c157	48	0	3 (2)	3	0	0	2	0	2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
c158	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	2
c159	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	2
c160	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	2

Table 10: **Number of Mutants for Classes #121 through #160**

classes	Lines	Veriables	Method (Constructors)	Inheritance depth	Mutants																						
					I HD	I H I	IOD	IOP	IOR	ISK	IPC	PNC	PMD	PPD	PRV	OMR	OMD	OAN	JTD	JSC	JID	JDC	EOA	EOC	EAM	EMM	Total
c161	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c162	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c163	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c164	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c165	29	0	2 (1)	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c166	51	1	3 (2)	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**2**
c167	114	0	8 (2)	2	0	1	4	1	4	0	1	20	0	0	0	0	0	0	0	0	0	0	0	0	3	0	**34**
c168	60	0	3 (2)	2	0	1	1	0	1	0	1	20	0	0	0	0	0	0	0	0	0	0	0	0	2	0	**26**
c169	31	0	1 (2)	3	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**
c170	31	0	2 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c171	75	2	8 (1)	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	**5**
c172	27	0	1 (2)	3	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**
c173	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c174	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c175	146	5	16 (1)	0	0	0	3	0	3	0	0	0	0	0	2	0	0	0	8	0	0	0	0	0	1	4	**21**
c176	62	0	3 (2)	3	0	5	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**11**
c177	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c178	31	0	2 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c179	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c180	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c181	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c182	27	0	1 (2)	3	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**
c183	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c184	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c185	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c186	713	13	56 (2)	2	0	0	1	0	1	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	**6**
c187	28	0	2 (1)	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c188	28	0	2 (1)	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c189	97	1	9 (2)	2	0	1	5	2	5	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	**15**
c190	49	0	3 (2)	2	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**2**
c191	74	1	7 (3)	1	0	2	3	0	3	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	**10**
c192	21	0	1 (1)	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c193	78	1	7 (1)	2	0	0	2	0	2	0	1	0	0	0	0	0	0	0	2	0	0	0	0	0	8	0	**15**
c194	24	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c195	24	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c196	137	1	3 (9)	0	0	0	1	0	1	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**5**
c197	51	0	3 (2)	4	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**
c198	50	0	3 (2)	4	0	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**
c199	94	2	8 (2)	1	0	2	3	0	3	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	**10**
c200	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**

Table 11: **Number of Mutants for Classes #161 through #200**

classes	Lines	Veriables	Method (Constructors)	Inheritance depth	Mutants																						
					I HD	I H I	IOD	IOP	IOR	ISK	IPC	PNC	PMD	PPD	PRV	OMR	OMD	OAN	JTD	JSC	JID	JDC	EOA	EOC	EAM	EMM	Total
c201	43	2	2 (2)	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	**4**
c202	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c203	25	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c204	65	1	6 (2)	1	0	1	3	1	3	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	**10**
c205	24	0	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**2**
c206	120	4	5 (2)	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	6	0	0	0	0	0	0	0	**7**
c207	68	0	3 (2)	3	0	5	2	2	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**11**
c208	24	1	1 (1)	2	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**
c209	152	5	4 (1)	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	3	0	1	0	0	0	6	0	**11**
c210	260	5	18 (1)	1	0	0	13	0	13	0	0	10	0	0	0	0	0	0	0	0	2	0	0	0	30	0	**68**
c211	215	3	12 (1)	1	0	0	3	0	3	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	46	0	**54**
c212	33	1	2 (1)	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**
c213	163	5	5 (1)	0	0	0	0	0	0	0	0	0	0	0	2	0	0	4	2	0	0	0	0	0	4	0	**12**
c214	116	4	4 (4)	1	0	0	1	0	1	0	0	0	0	0	1	0	0	0	5	0	4	0	0	0	5	0	**17**
c215	79	2	6 (1)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	9	0	**11**
c216	**319**	**5**	**14 (2)**	**0**	**0**	**0**	**3**	**0**	**3**	**0**	**0**	**0**	**0**	**0**	**0**	**8**	**0**	**12**	**1**	**3**	**0**	**0**	**0**	**0**	**6**	**0**	36
c217	37	1	4 (0)	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	**3**
c218	51	1	5 (0)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	12	0	**13**
c219	37	1	4 (0)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	**1**
c220	29	1	4 (0)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	**1**
c221	438	7	3 (1)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**
c222	163	7	3 (1)	0	0	0	0	0	0	0	0	0	0	0	4	0	0	7	4	0	0	0	0	0	9	0	**24**
c223	76	1	3 (2)	0	0	0	0	0	0	0	0	10	0	0	0	0	0	0	1	0	0	0	0	0	3	0	**14**
c224	111	4	2 (1)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**
c225	112	4	9 (1)	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	3	0	0	0	8	0	**13**
c226	52	1	1 (1)	0	0	0	1	0	1	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**4**
c227	73	1	2 (1)	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**2**
c228	70	9	4 (1)	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**
c229	171	5	8 (1)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**
c230	361	38	15 (1)	5	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**2**
c231	61	2	5 (1)	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**2**
c232	60	2	5 (1)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**
c233	396	11	21 (10)	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**
c234	50	1	3 (2)	3	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**2**
c235	21	0	0 (2)	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**1**
c236	16	0	0 (1)	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	3	9	**14**
c237	10	0	0 (0)	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	**1**
c238	21	0	0 (2)	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**1**
c239	21	0	0 (2)	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	**2**
c240	16	0	0 (1)	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**

Table 12: **Number of Mutants for Classes #201 through #240**

classes	Lines	Veriables	Method (Constructors)	Inheritance depth	Mutants																						
					I HD	I H I	IOD	IOP	IOR	ISK	IPC	PNC	PMD	PPD	PRV	OMR	OMD	OAN	JTD	JSC	JID	JDC	EOA	EOC	EAM	EMM	Total
c241	16	0	0 (1)	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	**1**
c242	21	0	0 (2)	6	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	0	**12**
c243	26	0	1 (1)	0	0	0	0	0	0	0	0	83	0	0	1	0	0	0	1	0	4	0	0	0	179	0	**268**
c244	27	1	1 (1)	1	0	0	0	0	0	0	0	7	0	0	0	0	0	0	2	0	0	0	0	0	153	0	**162**
c245	38	1	2 (1)	0	0	0	1	0	1	0	0	0	0	0	2	0	0	0	0	0	4	0	0	3	7	0	**18**
c246	65	2	6 (0)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	2	0	0	0	0	0	**3**
c247	48	1	2 (1)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**
c248	27	1	1 (1)	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	**1**
c249	64	2	3 (1)	1	0	0	2	0	2	0	0	0	0	0	5	0	0	0	2	1	0	0	0	0	0	0	**12**
c250	203	2	14 (1)	1	0	0	2	0	3	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	**9**
c251	195	4	6 (1)	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**0**
c252	**205**	**2**	**33 (1)**	**1**	**0**	**0**	**32**	**0**	**32**	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**65**
c253	66	4	4 (1)	0	0	0	2	0	2	0	0	0	0	0	0	1	0	1	2	0	0	0	0	1	0	0	**9**
c254	34	2	2 (1)	0	0	0	2	0	2	0	0	0	0	0	0	2	0	0	8	0	1	0	0	1	0	0	**16**
c255	47	1	1 (1)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	1	2	0	0	0	23	0	**30**
c256	1114	2	154 (1)	1	0	0	149	0	149	0	0	0	0	0	1	0	0	0	2	0	2	1	0	0	27	0	**331**
c257	73	3	6 (2)	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	**3**
c258	10	0	0 (0)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	**2**
c259	1906	4	175 (1)	1	0	0	163	0	163	0	0	0	0	0	24	0	0	0	3	1	4	1	0	0	75	0	**434**
c260	132	1	10 (1)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	**2**
c261	164	2	16 (2)	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	4	0	36	1	0	0	24	237	**305**
c262	186	3	3 (1)	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	1	0	0	0	0	**3**
c263	204	2	6 (1)	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	1	0	0	0	0	**3**
c264	36	1	2 (1)	2	0	0	0	0	0	0	10	0	0	0	0	0	0	2	8	0	10	0	0	0	132	118	**280**
total	**22,859**	**367**	**1,649 (430)**	**460**	**0**	**96**	**496**	**44**	**497**	**0**	**173**	**243**	**0**	**0**	**289**	**59**	**10**	**53**	**203**	**11**	**115**	**106**	**4**	**5**	**1,016**	**392**	**3,812**

Table 13: **Number of Mutants for Classes #241 through #264**