To appear in Proc. IEEE Int. Symp. Software Reliability Engineering (ISSRE) 2002.

# Mutation of Java Objects

Roger T. Alexander, James M. Bieman, Sudipto Ghosh, Bixia Ji
*Department of Computer Science*
*Colorado State University*
*Fort Collins, Colorado 80523*

## Abstract

*Fault insertion based techniques have been used for measuring test adequacy and testability of programs. Mutation analysis inserts faults into a program with the goal of creating mutation-adequate test sets that distinguish the mutant from the original program. Software testability is measured by calculating the probability that a program will fail on the next test input coming from a predefined input distribution, given that the software includes a fault. Inserted faults must represent plausible errors.*

*It is relatively easy to apply standard transformations to mutate scalar values such as integers, floats, and character data, because their semantics are well understood. Mutating objects that are instances of user defined types is more difficult. There is no obvious way to modify such objects in a manner consistent with realistic faults, without writing custom mutation methods for each object class. We propose a new object mutation approach along with a set of mutation operators and support tools for inserting faults into objects that instantiate items from common Java libraries heavily used in commercial software as well as user defined classes. Preliminary evaluation of our technique shows that it should be effective for evaluating real-world software testing suites.*

**Keywords:** *Faults, Java, mutation analysis, object-oriented programming, software testing, test adequacy, testability*

## 1. Introduction

Program testing is an integral part of software development processes. Testing a program involves the creation of test cases, the execution of the program against these test cases, and the observation of program behavior to determine correctness. A *test case* is a sequence of input values supplied to a program to test it, along with the expected output. A *test set* is a set of one or more test cases. A test case is successful if the observed behavior conforms to functional requirements; otherwise, the test fails. Success may be determined with the help of an *oracle* that compares the observed output with the expected (correct) output.

Test adequacy assessment is the evaluation of how thorough the testing was, and is an indicator of the goodness of the test sets. An adequacy criterion is defined as a predicate that defines what properties of a program must be exercised to constitute a thorough test [11]. The term *coverage domain* denotes a set of program related entities that are checked and counted for measuring coverage. Within a program these entities include functions, statements, decisions, and definition-use pairs. Test coverage is measured with respect to a particular coverage domain and reflects *how much* of that domain has been executed and tested.

Testability is defined by the IEEE Standard Glossary of Software Engineering Terminology (1990) as:

> "(1) the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met, and (2) the degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met."

Voas [21] defines testability as the likelihood of a program failing on the next test input from a predefined input distribution, given that there is a fault in the program.

Fault insertion based approaches have been used for measuring test adequacy [7, 5, 17] and testability [20]. Mutation analysis inserts faults into a program with the goal of creating mutation-adequate test sets that distinguish the mutant from the original program. Software testability is measured by seeding faults into a program. In both cases, inserted faults must represent plausible errors. It is relatively easy to apply standard transformations to mutate scalar values such as integers, floats and character data, because their semantics are well understood. Inserting faults into objects that are instances of user-defined types is more difficult. There is no obvious way to modify such objects in a manner consistent

with realistic faults, without writing custom mutation methods for each class. For example, one may define a mutation operator that nullifies an object reference. However, the operator is actually being used to change an object reference (which is scalar) and not necessarily the state of the object itself. The semantics of an object reference may be understood without knowing the semantics of the object itself. To mutate an object, one needs to know its semantics. Why is the ability to mutate user defined-types important? Because of the proliferation of user-defined types in software written using object-oriented (OO) languages.

Traditional mutation techniques use operators that are applied to constructs (variables, constants, operators and statements) in program code. These techniques cannot be easily extended to object-oriented programs where the variables usually represent objects (actually object references). Some OO mutation approaches use mutation operators that operate at code level to mutate aspects of attribute and method visibility, and inheritance. However, they do not address the important issue about object semantics.

In a prior paper, we introduced a new approach to object mutation that involves the definition of mutators (short for mutation operators) for classes of objects that need to be mutated [2]. We found that it is not necessary to define mutators for every class. We can mutate Java library items that are heavily used in commercial software. We can select applicable mutators based on (1) the inheritance hierarchy of classes and (2) the hierarchy of available mutators.

This paper further refines our approach to object mutation and describes our mutator operations and support tools. We have implemented an *Object Mutation Engine* that selects appropriate mutators and applies them to mutate objects. Our mutators are applied to objects at run-time, unlike other approaches where mutants are created from program code, compiled and then executed. Preliminary evaluation of our technique indicates that reusable libraries of mutation components can effectively insert faults into objects that instantiate items from the common Java libraries.

The remainder of this paper is organized as follows. We summarize background material on mutation analysis in Section 2. We describe our object mutation approach in Section 3 and explain the Object Mutation Engine in Section 4. We demonstrate our approach in Section 5. In Section 6 we present our conclusions and outline directions for future work.

## 2. Background on Mutation Analysis

Mutation analysis involves the modification of programs to see if existing tests can distinguish the original program from the modified program (also called *mutant*). The mutants must compile. Traditionally, syntactic modifications have been used and they are determined by a set of *mutation operators*. This set is determined by the language of the program being tested and the mutation system used. Mutation operators are created with one of two goals: (1) to induce simple syntax changes based on errors that programmer typically make (e.g. use a wrong variable name), and (2) to force common testing goals (e.g. execute each branch).

Mutation testing relies on the *Competent Programmer Hypothesis* and the *Coupling Effect*. The *Competent Programmer Hypothesis* states that programmers are generally competent and produce a program close to a correct program. A correct program can be constructed from an incorrect program by making changes that are composed of minor alterations. The *Coupling Effect* states that test cases that distinguish programs with minor differences from each other are so sensitive that they can distinguish programs with more complex differences. The competent programmer hypothesis and the coupling effect imply that small changes in programs are adequate to help reveal complex errors.

An example of a programmer error in a C program is typing $x > y$ instead of the intended $x >= y$. Thus, one may define a mutation operator that takes an expression containing $>=$ and replaces it with $>$. The goal for the tester is to create a test case that can detect a difference in the outputs of the program containing $>=$ and the mutant containing $>$. Other examples of mutation operators include (1) replace a binary operator by another, (2) replace a variable by another variable used in the program, and (3) replace a constant by another constant. Mutation operators were defined for procedural programming languages such as Fortran [6] and C [1, 3, 4].

Let $P$ be the program under test and $P_c$ be one correct version of $P$. If $P$ is correct, $P$ and $P_c$ are the same. $\mathcal{T}$ is the set of tests used to test $P$. Let the input domain of $P$ be denoted by $\mathcal{D}$. Mutation testing relies on a set of faults $\mathcal{F}$. Each fault $f$ in $\mathcal{F}$ is introduced in $P$ one by one. Introduction of a fault into $P$ results in a program $M$ that is slightly different from $P$. The program $M$ is called a *mutant* of $P$. The application of all the faults in $\mathcal{F}$ one by one into $P$ produces a set of mutants $\mathcal{M}$. Elements of $\mathcal{F}$ are known as *mutation operators*. When a mutant $M$ is executed against a test case $t$ in $\mathcal{T}$ and the behavior of $M$ is different from that of $P$, the mutant $M$ is said to be *killed* by $t$. A tester is expected to kill each mutant in $\mathcal{M}$ with at least one test case $t$. In case a mutant cannot be killed, the tester needs to show that $M \equiv P$ or update $\mathcal{T}$ by adding a test case $t' \notin \mathcal{T}$ that kills $M$. A mutant that is not killed during testing is said to be *live*. The adequacy of $\mathcal{T}$ is determined by the ratio of the number of mutants killed to the number of non-equivalent mutants in $\mathcal{M}$. This ratio is also called the mutation score. $\mathcal{T}$ is considered adequate if the mutation score is 1.

### 2.1. Mutation Operators for Object-Oriented Programs

Mutation operators defined for procedural languages such as Fortran and C are also applicable to Java. However, Java has several additional features that arise out of the OO programming paradigm. Offutt et al. [18] applied mutation analysis to Ada programs and addressed some of the object-oriented features, but the analysis was limited to properties within a class and did not address inheritance.

Kim et al. [12, 13] used Hazard and Operability Studies Analysis (HAZOP) to define mutation operators for Java. HAZOP was applied to the Java syntax definition to identify deviations of Java language constructs. The analysis provided a list of Java mutation operators and helped create a database of flaws for Java programs. The operators apply to types and variables, modifiers, class and interface declarations, blocks and expressions. Specifically, class mutation targets faults related to the OO-specific features of Java: (1) class declarations and references, (2) single inheritance, (3) information hiding and (4) polymorphism. The proposed Java mutation operators represent "generic" faults that do not take object semantics into account. Several operators listed by Kim et al. will result in mutants that do not compile. The issue of integration testing is not addressed. While there are some operators that mutate parameters of methods to model the faults related to erroneous use of overloaded methods, there could be other sources of integration errors.

An object possesses state and makes transitions from one state to another. Mutation operators that are applied to program code are not effective in ensuring that objects will go through different states. The necessity for testing of objects in different states has been shown in several testing techniques: Doong and Frankl [8], Kirani and Tsai [14] and Kung et al. [15].

### 2.2. Mutation Analysis for Integration Testing

Delamaro et al. [4] first described the technique of *interface mutation* for the integration testing of C programs. The underlying idea is to create mutants by inducing simple changes only in the entities belonging to the interface between modules or sub-systems. The technique is designed to be scalable with the size of the software under test. The size is measured as the number of sub-systems being integrated. To address the mutant explosion problem normally associated with traditional mutation techniques, the interface mutation approach (1) restricts the mutation operators to model only integration errors, (2) tests only the connections between two modules, a pair at a time, and (3) applies the integration mutation operators only to module interfaces such as function calls, parameters or global variables.

Ghosh and Mathur [10] applied interface mutation analysis to distributed object systems. They defined interface mutation operators based on interface descriptions that described methods and exceptions. Operators would be applied to method parameters and return values. Testers would mutate the *client* program's method call or the definition of the method implementation in the *server*. A server mutation affects all the callers of the mutated method.

The mutation operators for interface mutation as described above can be easily defined and applied when the parameters are scalars. However, mutating parameters that are objects is more difficult. State mutation operators cannot be applied statically to a program, because the state of the object depends on program execution. In addition, the interface mutation operators cannot represent state errors caused by specific sequences of method execution. Such errors depend on the order of operations and not necessarily on the values of the arguments.

## 3. Approach

A goal of our work is to identify plausible faults that can be injected into objects to put them in faulty states for instantiations of arbitrary user-defined types. Identifying mutation operators is difficult. We begin by analyzing the Java Application Programming Interface (API) and identify mutation operators that apply to instantiations of classes defined in the API.

Instead of defining operators for each class, we define operators that can apply to a whole group of classes that implement a certain interface. For example, we look at mutation operators (mutators) that apply to the following interfaces:

1. Container types defined in the interfaces, *Collection* and *List* in the package `java.util`.
2. Iterators defined in the interface *Iterator* in the package `java.util`.
3. InputStream defined in the abstract class *InputStream* in the package `java.io`.

```
1.        class C {
2.            ...
3.            public void m (Foo f) {
4.                ...
5.                ...
6.            }
7.            ...
8.        }
```

**Figure 1. Code of Class C**

The code in Figure 1 shows a class `C` containing a method `m()` which takes a parameter `f` that references an object of type `Foo`. The code inside `m` is not shown. The code uses the

parameter `f` for computation. We wish to mutate the object bound to `f` before it is used. This will involve inserting just after line 3, the statement shown below.

```
f = (Foo) ObjectMutationEngine.mutate(f);
```

We can also mutate an object returned by a method as shown in Figure 2. To do this, we place the `mutate` call just prior to the `return` statement. The ObjectMutationEngine implements the `mutate` methods. Inserting calls to the ObjectMutationEngine is relatively easy using a code instrumenter that builds a parse tree and instruments certain nodes in the tree with the `mutate` method.

```
1.        class C {
2.            ...
3.            public Bar m (Foo f) {
4.                ...
5.                Bar b;
6.                ...
7.                return b;
8.            }
9.            ...
10.        }
```

**Figure 2. Mutation of Return Statement**

The details of the ObjectMutationEngine are described in Section 4.

### 3.1. Default mutators

For an instance of an arbitrary user-defined type we look at the fields of the object. The Java reflection API enables us to identify the types of objects. This applies to `public`, `private` and `protected` fields. If the field is a scalar, we apply the traditional scalar mutators as follows:

1. Increment the value by 1
2. Decrement the value by 1
3. Set the value to a constant

For fields that are objects we use our new mutators. If we know the semantics of the objects, it is easier to select the mutators. In case we do not know the semantics, certain default mutators can be applied. A default mutator could be one that makes an object reference `null`. This mutation may not be useful as it will probably raise a `NullPointerException`. Moreover this is an operation that is applied to the reference, not the object. This mutation treats a variable that refers to an object as having an underlying type of "reference to object" and takes advantage of knowledge of the semantics of variables of this sort. All that it can do with a variable that is an object reference is take its value, assign it a value, test for equality between two such variables (i.e. do they refer to the same object), and dereference its value (i.e. using the member access operator).

Another mutator for arbitrary object types is one that recursively applies mutators to the nested fields until the scalar fields are reached. The following mutators can also be applied:

1. Cloning the object referred to by the variable and assigning the reference to the clone to the variable. For a testing perspective, this would test the sensitivity of a program to the identity of an object as opposed to the state of the object.

2. Creating a new object whose type is compatible with the declared type $T$ of the object reference. In other words, we can instantiate a new object whose type is a descendant $D$ of $T$. Options here would be to have the state variables common to both $D$ and $T$ be the same. That is, the two objects would be the same with respect to their common ancestors. They could be the same from either an equality perspective (i.e. shallow copy), or from the perspective of equivalency (i.e. deep copy).

### 3.2. Mutators for containers

Java provides several container interfaces, such as `Collections` and `Iterators`. Classes that implement these interfaces are also provided as part of the API. We identify mutators for these interfaces and classes using their semantics.

#### 3.2.1 Mutators for the Collection interface

The following mutators are implemented for the `Collection` interface.

1. **Make the Collection empty:**

   In Java every `Collection` needs to implement the method `clear()`. The mutate method for making the `Collection` empty would just invoke the `clear()` method on this object. This mutator models errors related to inserting elements in an unintended collection, or accidental removal of elements from a collection.

2. **Remove an element from the Collection:**

   The element to be removed from the `Collection` could be the first, last or some random element. Every `Collection` provides a `remove()` method that takes an object as a parameter. We can select the first, last or random object by using the `toArray()` method on the `Collection` and getting an array of Objects. We can index this array and select any object to be removed. This mutator models errors that occur because the programmer forgot to put an element inside the collection (may be an off-by-one error).

3. **Add an element to the Collection:**

   We can add some arbitrary element to the `Collection` using the `add()` method provided in any `Collection`. This element could be a clone of some existing element. Alternatively, the element constructor that takes no parameters or the default constructor of the element, if available, can be used to create a new element. This mutator models redundant additions of elements into a collection.

4. **Mutate the elements:**

   For every element inside the `Collection`, the mutate method for the element's type can be invoked. This mutator is used to recursively apply mutation to objects.

5. **Reorder $m$ of the $n$ elements in the Collection:**

   A collection may be implemented using a class that makes use of ordering. This mutator models errors that may be made in the ordering. Since there is no notion of order embodied in a `Collection`, and there is no method provided in the API that can manipulate the order, we cannot apply the reorder mutator to the `Collection` interface directly. We can still use the method `toArray()` to obtain an array of the objects, reorder the array and create a new `Collection` from the objects in the array.

#### 3.2.2 Mutators for the Iterator interface

The `Iterator` interface provided in the Java API allows one to obtain the next element in the iteration using the method `next()`. We define a **skip** mutator that makes the iterator *skip* elements. The **skip** mutator is implemented by a `mutate` method that calls the `next()` method one or more times.

This mutator does not affect all instances of the mutated type but just the instance held by some client. Since we are only mutating one iterator and not the original container, any other client using the container or a different iterator over the container will not see a difference. This mutator is similar to the **Remove element** mutator except that it must be detected by testing code that deals with the iterator and not the collection itself.

#### 3.2.3 Mutators for List and Vector

The mutators defined for the `Collection` and `Iterator` interfaces apply to the `List` interface as well. The faults they model are similar as well. In addition, the `Collections` class provides a number of static methods, such as `shuffle(List list)` and `shuffle(List list, Random rnd)` that can be used to reorder the elements in the `List`.

The specific semantics of implementations can be used to mutate container objects. For example, a binary tree has a notion of ordering. This notion can be used to implement a **reorder** mutator. The following mutators are implemented for `List` and `Vector`:

1. **Randomly reorder** the elements in the `List`.
2. **Delete** the first/last/any element from the `List`.
3. **Delete** the first/last/any element from the `Vector`.

### 3.3. Mutators for map

The following mutators are implemented for `Map`.

1. **Delete** the first mapping.
2. **Delete** the last mapping.
3. **Empty** the map.

### 3.4. Mutators for inputstreams

The `InputStream` abstract class in the Java API provides a method called `skip(long n)` which skips over and discards `n` bytes of data from this input stream. We define a mutator for inputstreams that results in the skipping of bytes of data. This mutate method will call the `skip()` method with an appropriate length parameter.

### 3.5. Mutators for user-defined classes

We implemented mutators for two user-defined classes `Region` and `GraphEdge` that were written for two applications, (1) *Paint* and (2) *GraphPath* respectively.

The mutators for `Paint` are:

1. **Increment_by_one** the height of a parent.
2. **Increment_by_one** the width of a region.
3. **Increment_by_one** the width and height of a region.

The mutator for `GraphPath` is:

1. **Increment_by_one** the number of graph edges.

## 4. Architecture of Mutation Engine

The abstract components of the Object Mutation Engine (OME) architecture are depicted in Figure 3. The parallelograms represent separate threads of control, the rectangles represent sequential processes that are executed by a thread, and the circles represent data. The dashed directed line segments between the threads or sequential processes represent control flow. The solid line segments superimposed with a circle represent both control flow and specific types of data flow. This architecture is realized by a Java implementation that provides abstractions and concrete types for each architectural component.

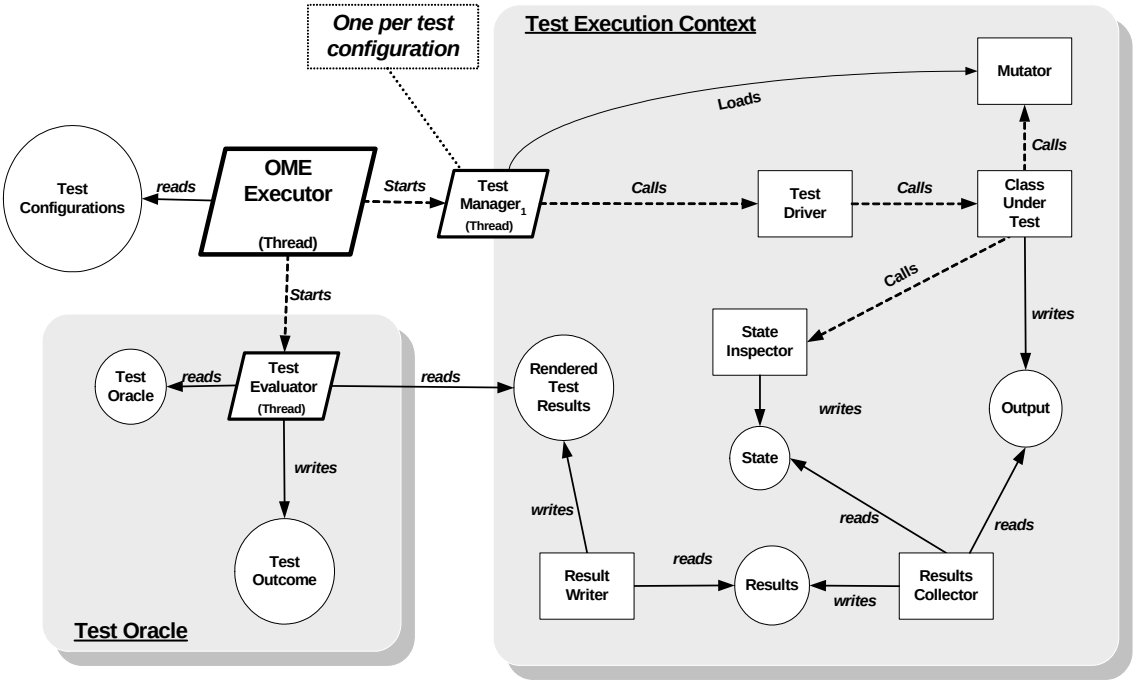


**Figure 3. Architecture of the Object Mutation Engine**

As the threads depicted by Figure 3 suggest, the OME is a concurrent system following a master-slave model of computation. It is envisaged that the OME will ultimately become a distributed system whereby individual tests will be parceled out to different nodes of a network. The objective is to reduce the amount of time required to conduct mutation testing, thereby overcoming one of its perceived weaknesses.

### 4.1 Execution of the OME

The execution of a single test within the OME begins when the *OME Executor* reads a *test configuration*. A test configuration contains parameters that describe the characteristics of the *Class Under Test* (CUT). This includes the fully qualified name of the CUT, the fully qualified name of the test driver to use, the number of locations within the CUT where a mutation operation will be applied, and the set of mutation operators that will be applied.

Upon reading a test configuration, the *OME Executor* creates a new *Test Manager* that will configure and run the test. The *Test Manager* runs on its own thread of execution. When created, it first establishes the environmental conditions necessary to execute the test. This includes loading the mutation operators that will be used during the test, and loading the *State Inspector* that will be used to capture state of the CUT and other objects. The *Test Manager* then loads the *Test Driver* that is specific to the CUT and that will execute the actual test.

When executed, the *Test Driver* performs initializations that are specific to the test and to the CUT. For example, this might include initializing database connections, staging test data, and so on. After completing initialization, the *Test Driver* creates the instance of the CUT that will be tested, and then executes the test. This is accomplished by making calls into the CUT instance that are necessary to carry out the test. Upon conclusion of the test, the *Test Driver* performs any cleanup actions necessary and then terminates, returning control to the *Test Manager*. The *Test Manager* then signals completion of the test back to the *OME Executor* and then terminates.

### 4.2 Variation within the testing process

One of the key objectives of the OME is to provide a general purpose framework that can support mutation testing of object-oriented programs. To accomplish this, the OME takes into consideration the variation that must necessarily occur due to syntactic and semantic differences in classes. There are five areas in which this variation may occur:

1. *Variation in the class C that is the subject of the test*: C is referred to generically as the *Class Under Test* (CUT). The syntactic interface of *C* and *C*'s behavior

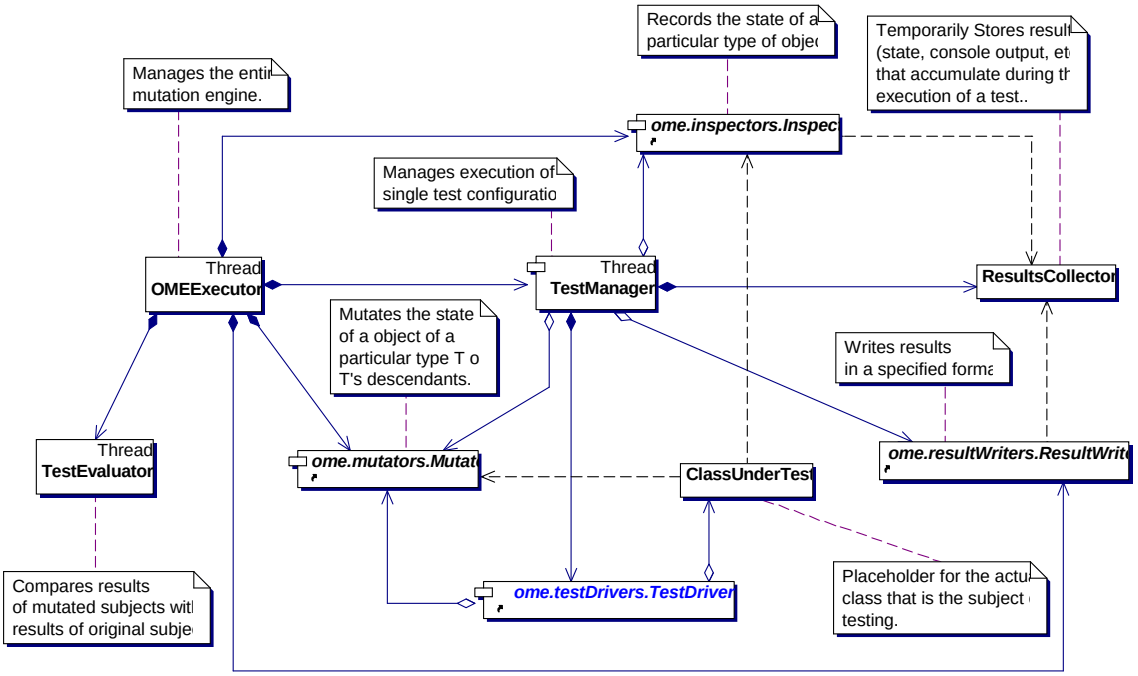


**Figure 4. UML Class Diagram of OME**

are unpredictable. Consequently, it is not possible to write a generic framework that can make any assumptions about the actual classes that will be tested.

2. *Variation in the driver that executes C under the conditions of a test*: Because of difficulties in (1), it is not possible to write a generic test driver that can establish the conditions necessary to execute a test of *C*.

3. *Variation in the operators that are used to mutate the state of objects that occur during the course of a test*: This too is a consequence of (1). Because the semantics of *C* can specify any behavior, the operations used to mutate instances of *C* cannot be predicted. Thus, it is not possible to automatically generate mutations without prior knowledge of the required operations and their semantics.

4. *Variation in the mechanism that is used to capture the state of C and related objects*: *C* is a user-defined type and has a concrete representation that is the foundation of its implementation. Part of this representation will likely include instances of other types, some of which will be scalars (e.g. integers and floating point numbers). and others will also be user-defined types. Because of the wide variation of types available for use, it is not possible to know in the general case the form in which to capture the internal state of *C*. For scalar types, a string representation is straightforward, but not for user-defined types. For example, what should the form be that is used to capture the value (state) of an instance of a carburetor?

5. *Variation in the mechanism that collects the output produced by C during the execution of a test*: Ultimately, programs are written for the output that results from their computation. This output has the potential to take an infinite number of representations. In the simplest case, the output can simply be text written to a console. In more complicated cases, the output could be something in the file system, such as the creation of a new file or modification to an existing file. More complicated cases include modifications to relational databases, or output to a graphical user interface.

The OME accounts for the variation described using mechanisms that isolate and abstract away the physical details of any particular *C*. The following subsections describe each of these mechanisms and the other components of the OME in detail.

### 4.3 Components of the OME

As noted at the beginning of section 4, the OME is implemented using the Java programming language. Figure 4 depicts a UML class diagram that illustrates a portion of the static structure of the OME's implementation. In particular, the diagram shows the key abstractions that comprise the

implementation, and their relationships. The following subsections describe each of these abstractions in detail.

#### 4.3.1 OME Executor

As depicted in Figure 3, the *OME Executor* plays the role of a master process that controls the execution of tests. Within the implementation of the OME, it is represented by the class *OMEExecutor*, as shown in Figure 4. The primary responsibility of this class is the loading of test configurations and creating corresponding test jobs that will manage the testing activities. To accomplish this, the *OMEExecutor* creates an instance of *TestManager* (section 4.3.2) and delegates to it the responsibility for overseeing the execution of a particular test job.

The *OMEExecutor* also is responsible for initiating one or more processes that will provide a testing oracle function. This is done by creating instances of *TestEvaluator*, each running on separate threads of control. Upon completion of a test job, the *OMEExecutor* assigns the responsibility for evaluating the results of the testing activities to an instance of *TestEvaluator*.

#### 4.3.2 Test Manager

As described in section 4.3.1, instances of *TestManager* oversee the activities associated with a particular test job. A test harness establishes any necessary environmental conditions required to support a test, loading the proper instance *T* of *TestDriver* associated with the CUT, providing a mechanism for collecting test results (*ResultCollector*), and making the mechanisms for state inspection (*Inspector*) and the rendering of results (*ResultsWriter*) available to *T*.

Instances of *TestManager* must also know which mutation operators will be applied during a particular test along with the number of times each mutation operator must be applied (i.e. the number of locations within the CUT). This information is passed from *OMEExecutor* as part of the test configuration.

#### 4.3.3 State Inspectors and Results Collection

One of the general problems with testing is that of capturing the results that must be used to determine the outcome. This problem is worse for OO software due to the decreased observability that results from encapsulation information hiding [19]. A complicating aspect of this problem is that output can occur in many forms, such as direct console output, changes to a local file system (e.g. creating/updating/deleting files and directories), updates to databases. In general, there is a plethora of different mechanisms that can produce output. Thus, it is not possible to write a general purpose mechanism that can capture all forms of output.

To overcome part of these problems (at least for software written in Java), the OME has a family of mechanisms that examine the state of arbitrary types of objects. This family is defined by the class *Inspector* shown in Figure 4. Each class in this family is capable of inspecting the state of a specific type of object. For example, the OME includes the class *DefaultInspector* that provides the capability of inspecting the state of instances of the most general type *java.lang.Object*.

When a field *f* is inspected, the type of the object bound to *f* is used to determine the specific kind of inspector to use. Thus, each type of inspector is written to inspect the state of a specific type of object (referred to as the inspector's *characteristic type*). The inspector that is selected is the one whose characteristic type is the closest in an inheritance hierarchy to the type of the instance bound to the field. Thus, a given inspector may be applicable to a number of different types of objects (specifically those that are instances of descendants of the inspector's characteristic type).

If the type of a particular field *f* in an object being inspected does not match an existing inspector, an instance of *DefaultInspector* is used to capture *f*'s value. If *f* is of a type unknown to *DefaultInspector*, its value is rendered using the *toString* provided by the class *java.lang.Object* (this class is the parent of all other user-defined types in Java). The OME also includes a predefined class *StringInspector* that is used to capture the state of instances of this type.

Instances of *Inspector* populate an instance of *ResultCollector* with the values of the state variables of an arbitrary object *o*. The Java Reflection API is used to gain access to o to query its individual fields for their values. These values are rendered as strings and stored as key/value pairs in an instance of *ResultCollector*. Ambiguities are avoided by having the keys structured as stylized names that include the fully qualified names of the state variables contained in the definition of the actual type of the object bound to *o*. The keys also embed the type of the field, as illustrated by: *java.lang.System.out[java.io.PrintStream]*

Inspectors are used during the testing process by calls embedded in the CUT at locations where the recording of intermediate and final state is required. Typically, the latter will be performed just prior to where control returns from a method that is executed as part of the test.

For capturing other forms of output, specialized types of *ResultCollector* can be written that know how to capture the various forms of output that a given CUT produces. The details of this vary widely from one CUT to the next. The default *ResultCollector* within the OME has the capability of capturing I/O written to the standard output and standard error devices. This information is buffered in the local file system until the final results of the test are rendered (section 4.3.4).

#### 4.3.4 Results Rendering

Results may be rendered from a given test in a number of different ways. The mechanism (*ResultsWriter*) employed by the OME allows the definition of custom mechanisms that can render test results in any arbitrary form. By default, results are rendered as simple textual representations of the key/value pairs contained in an instance of *ResultCollector*. The OME also includes a specialized *ResultCollector* that renders test results and capture output as an XML document. This is then used by the *TestEvaluator* to determine the test outcome.

#### 4.3.5 Test Drivers

Each CUT *C* represents a type defined by a distinct syntactic unit that has a unique semantics. A consequence of this is that writing a general purpose test driver that can be applied to any *C* is impractical. Such a test driver *D* would be required to have the capability to establish the environmental conditions for any *C*, and be able to create instances of *C* for testing. Given that the set of all *C* is potentially infinite, writing *D* would not be possible.

Recognizing that all test drivers can be considered the same from a very abstract perspective, it is possible to define a common interface and associated behaviors that each *D* would possess. For example, each driver must initialize environmental conditions, load test data, execute one or more tests, and record results. While the specifics of these activities are likely to be different for any two test drivers, all test drivers can be assumed to have these activities and execute them in the same order. This assumption underlies our *TestDriver* abstraction contained within the OME which utilizes a *Template Method* design pattern [9] to define the algorithm for executing a test that all test drivers use. Essentially, this algorithm consists of steps that first perform whatever actions are necessary to setup for a test, runs the test, cleans up, and finally records results. Each step is delegated to an abstract method that each concrete test driver must implement. Thus, generality in the test driver is achieved by taking advantage of inheritance.

Each concrete test driver has several responsibilities within the OME that must be satisfied. The driver must know how to create the environmental conditions necessary to execute its associated test. This could include creating files, database connections, and so on. The driver must also know where to get its test data from. This could, for example, be defined within the driver itself, or could exist in a file or external database. Perhaps the most important responsibility the driver has is in creating the instance of the CUT that will be used as the test subject.

#### 4.3.6 Mutation Operators

The source code of the CUT is instrumented with calls to mutation operators (and to the instance of *Inspector*). Calls to mutation operators are inserted within the methods of the CUT at locations that involve objects that are of interest from a testing perspective. Typically, these will be immediately before or after locations that have calls to methods or statements that reference the state of a particular object in some manner. The particular mutation operator used is determined by the declared type of the object reference used at a particular location of interest. The basic idea is to create a special mutator class for each type or family of types that need to be mutated. These mutator classes implement a mutate method to support fault models relevant to a particular type — interface or class — and make use of the underlying semantics of the type.

Each mutation operator is implemented as a class that is a descendant of the Mutator class, shown in Figure 4. This class specifies the method `mutate` that takes a reference to an instance of java.lang.Object. The *characteristic type* of a mutator is that of the actual type of the mutation candidate (i.e. the reference that is bound to an object that will be mutated). Similar to the use in inspectors (section 4.3.3), the characteristic type is used to identify the types of object that the mutation operator is compatible with. Thus, generality for mutation is achieved by allowing the definition of custom operations that are specific to particular types.

#### 4.3.7 Test Evaluation

Once a test is complete, as described in section 4.3.1, the *OMEExecutor* calls on an instance of *TestEvaluator* to determine the outcome of a particular test. *TestEvaluator*, shown in Figure 4, is an abstract class that defines a set of common behaviors and interfaces. This allows for the definition of custom test evaluation mechanisms. This is necessary to achieve generality since the differences in test results and output will have a high degree of variation.

## 5. Demonstration of OME

We used the OME to mutate a selection of Java programs. The applied mutations include a set of mutators designed to mutate both standard Java library objects and custom mutators that can inject faults into user defined class objects. Only a small set of mutators were implemented for this initial demonstration of the OME. The standard library mutators include the following:

- in ome.mutators.java.util:
  - CollectionMutator: seven mutate methods delete or add an object from or to a Collection, or change the order of the objects in a Collection.

**Table 1. Applying the OME to five example programs.**

| Program | Number of Potential Locations | Num Actual Mutated Locations | Number of Mutators Per Location | Number of Tests | Number of Mutants | Number of Mutants Killed |
|---|---|---|---|---|---|---|
| Ant | 21,065 | 16 | 1–7 | 2 | 53 | 28 |
| Junit | 1,478 | 4 | 6 | 1 | 24 | 16 |
| GraphPath | 142 | 6 | 1–7 | 8 | 21 | 10 |
| Paint | 323 | 4 | 3 | 13 | 12 | 8 |
| MazeGame | 1,624 | 6 | 3 | 6 | 18 | 12 |

  - ListMutator: two mutators — one deletes the last element in a List, the other changes the order of List elements.
  - VectorMutator: deletes the last object in a Vector.
  - MapMutator: three mutators — one deletes the first element, one deletes the last element, and one makes a Map empty.
  - IteratorMutator: skips the next object.

- in ome.mutators.java.io:
  - InputStreamMutator: skips a random number of bytes in an input stream.

We applied the library mutations and some custom mutations to classes in a selection of Java programs including open source programs and programs used for classroom examples and assignments:

- Ant: a build tool from Apache. We mutated class org.apache.tools.ant.Project in version 1.4.1 of jakarta-ant. Three library mutators were applied: CollectionMutator, MapMutator, and InputStreamMutator.
- Junit: a unit testing framework written by E. Gamma and K. Beck. We mutated class junit.framework.TestSuite in Junit 3.7. CollectionMutator was applied.
- GraphPath: finds the shortest path and distance between specified nodes in a directed graph using Dijkstra's shortest path algorithm. CollectionMutator, IteratorMutator, and InputStreamMutator were applied. In addition, a custom mutator modified the length of a graph edge by one.
- Paint: calculates the amount of paint needed to paint a house. This program is adapted from a text by Lambert and Osborne [16]. A custom mutator modifies a painting region by adding one to the height and/or width.
- MazeGame: a game that involves finding and rescuing a hostage in a maze. This was a development assignment in a software engineering course. MapMutator was applied.

Table 1 shows the number of locations that were mutated and the number of mutators that were generated for each program. Only half of the 128 mutant programs were killed by our test data. We do not claim that our test data was comprehensive — we did not generate a large number of test cases. However, only a few test cases were needed to kill half of the mutants. It is also clear that the mutations represent plausible errors — the mutants had some chance of surviving the testing process, since half of the mutants remained alive.

This demonstration represents only a small fraction of all possible mutations for the selected programs. We mutated only a few locations in one class for each program. This is especially obvious in Ant where we mutated only 16 out of 21,065 possible locations, where each possible location is where a method is called. We leave the problem of how to optimally select locations to mutate as future work. Our objective here is to demonstrate that our methods for mutating objects can be applied to real programs.

## 6. Conclusions and Future Work

We described a technique for performing mutation analysis on object-oriented programs by injecting faults into objects. Our techniques make mutation work for OO software.

We showed that reusable libraries of mutation components can effectively inject plausible faults into objects that instantiate items from common Java libraries as well as user defined classes. Since Java library items are heavily used in commercial software, our technique should be effective for evaluating the real-world software testing suites. We can also develop custom mutators for user-defined types.

We designed an object mutation engine that implements our technique. We are now assessing the effectiveness of this technique. The design of the Object Mutation Engine is flexible. It can inject a wide variety of mutations

into running programs, and can be extended to support new fault models. With our techniques and tools, mutation of component-based systems is also possible.

Our next task is to experimentally evaluate the object mutation technique. We are investigating the cost of using the mutation framework and the gains of using our mutation technique. From these experiments we will have a better understanding of several issues — scalability, relative fault detection capabilities of test suites created using different sets of mutators, and the adaptability of the mutation framework to different application types. We will also be able to develop a set of guidelines that will aid the tester in using the object mutation technique on a particular application.

## References


[1] H. Agrawal, R. DeMillo, R. Hathaway, W. M. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. Mathur, and E. Spafford. Design of Mutant Operators for the C Programming Language. Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana, March 1989.

[2] J. Bieman, S. Ghosh, R. Alexander. A technique for mutation of Java objects. *Proc. Automated Software Engineering (ASE 2001)*. 2001.

[3] M. E. Delamaro, J. C. Maldonado, M. Jino, and M. L. Chaim. PROTEUM: Uma ferramenta de teste baseada na analise de mutantes (proteum: A test tool based on mutation analysis). In *Software Tools Proceedings of the VII Brazilian Symposium on Software Engineering*, October 1993.

[4] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Integration Testing Using Interface Mutation. In *Proceedings of International Symposium on Software Reliability Engineering (ISSRE '96)*, pages 112–121, April 1996.

[5] R. A. DeMillo, D. S. Guindi, K. N. King, and W. M. McCracken. An Overview of the Mothra Software Testing Environment. Technical Report SERC-TR-3-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana, August 1987.

[6] R. A. DeMillo et al. An Extended Overview of the MOTHRA Testing Environment. In *Workshop of Software Testing, Verification and Analysis*, July 1988.

[7] R. A. DeMillo and A. J. Offutt. Constraint-based Automatic Test Data Generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[8] R.-K. Doong and P. G. Frankl. The ASTOOT Approach to Testing Object-Oriented Programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, April 1994.

[9] E. Gamma, R. Helm, J. R., and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading MA, 1995.

[10] S. Ghosh and A. P. Mathur. "Interface Mutation". *Journal of Software Testing, Verification and Reliability*, 11(4):227–247, December 2001.

[11] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, 1975.

[12] S. Kim, J. A. Clark, and J. A. McDermid. "The Rigorous Generation of Java Mutation Operators Using HAZOP". In *Proceedings of the 12th International Conference on Software and Systems Engineering and their Applications (ICSSEA'99)*, Paris, France, December 1999.

[13] S. Kim, J. A. Clark, and J. A. McDermid. "Class Mutation: Mutation Testing for Object Oriented Programs". In *Proceedings of the FMES 2000*, 2000.

[14] S. Kirani and W. T. Tsai. "Method Sequence Specification and Verification of Classes". *Journal of Object-Oriented Programming*, pages 28–38, October 1994.

[15] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. "Change Impact Identification in Object Oriented Software Maintenance". In *Proceedings of IEEE International Conference on Software Maintenance*, pages 202–211, 1994.

[16] K. Lambert and M. Osborne. *Java: Complete Course in Programming and Problem Solving*. ITP Press. 1998.

[17] A. P. Mathur and W. E. Wong. A Theoretical Comparison Between Mutation and Data Flow Based Criteria. In *Proceedings 22nd Annual ACM Computer Science Conference on scaling up: meeting the challenge of complexity in real-world computing applications*, pages 38–45, Phoenix, Arizona, USA, March 8-10 1994.

[18] A. J. Offutt, J. Voas, and J. Payne. Mutation Operators for Ada. Technical Report ISSE-TR-96-09, Information and Software Systems Engineering, George Mason University, Fairfax, Virginia, March 1996.

[19] J.E. Payne, R.T. Alexander, and C.D. Hutchinson. Design-for-Testability for Object-Oriented Software. *Object Magazine*. pp. 34-43, 1997.

[20] J. M. Voas and G. McGraw. *Software Fault Injection: Innoculating Programs Against Errors*. John Wiley and Sons, January 1998.

[21] J. M. Voas and K. W. Miller. "Software Testability: The New Verification". *IEEE Software*, pages 17–28, May 1995.