**Automated Mutation Testing Framework**

**Project Mid-term Progress Report**

Zengtai Qi

The University of Texas at Dallas  
800 W. Campbell Road,  
Richardson, Texas 75080-3021  
+1 (469) 642-3217

Zengtai.Qi@utdallas.edu

**ABSTRACT**

In this paper, I describe the specific details of how the automated mutation testing is goanna work. I will introduce the approaches of mutant generation, the approaches of test selection, the approaches of mutation insertion and the approaches of mutation detection. Also I will explain which approach I use or plan to use in this project. Besides, I will introduce the mutators I plan to implement in this project.

**CCS Concepts**

**Software and its engineering → Software creation and management → Software verification and validation → Software defect analysis → Software testing and debugging**

**Keywords**

Mutation testing; weak mutation; strong mutation; JUnit.

# INTRODUCTION

The Mutation testing is conceptually quite simple1.

Faults (or mutations) are automatically seeded into code, then the tests are run. If the tests fail then the mutation is killed, if the tests pass then the mutation lived.

The quality of your tests can be gauged from the percentage of mutations killed.

To put it another way – the automated mutation testing framework runs a program’s unit tests against automatically modified versions of the program. When the program code changes, it should produce different results and cause the unit tests to fail. If a unit test does not fail in this situation, it may indicate an issue with the test suite.

Traditional test coverage (i.e line, statement, branch etc) measures only which code is executed by your tests. It does not check that your tests are actually able to detect faults in the executed code. It is therefore only able to identify code that is definitely not tested.

The most extreme example of the problem are tests with no assertions. Fortunately, these are uncommon in most code bases. Much more common is code that is only partially tested by its suite. A suite that only partially tests code can still execute all its branches.

As it is actually able to detect whether each statement is meaningfully tested, mutation testing is the gold standard against which all other types of coverage are measured.

# MUTANT GENERATION

## Source Code

Source mutators create mutations by making changes to the Java source files and recompiling them.

In this way, a large range of mutations can be generated, mutations can closely mimic the types of error a programmer might make and the mutations made can be clearly described and nderstood.

However, source based mutators are generally harder to integrate into a build, generating mutations in this way is relatively slow, mutants must be written to disk, limiting the methods by which they can be inserted and in theory a mutant class could be accidentally released.

## Byte Code

Byte code mutators create mutations by manipulating the compiled byte code. This will usually be done using a third party library such as ASM, javassist or BCEL.

Manipulating byte code, is generally much faster and generally easier to integrate into a build, can potentially create mutants without access to source files and same mutation operators can in theory work for other JVM languages.

However, it is more difficult to develop mutation operators this way. Also, mutations are divorced from the source code and may not be representative of errors a programmer would make. And it can be hard to describe / explain the mutations that must work around intricacies of the JVM.

## My Choice

In this project, I chose to manipulate the compiled byte code to fulfill mutant generation jobs. The tool I used is ASM framework. ASM is an all purpose Java bytecode manipulation and analysis framework2.

# TEST SELECTION

There are great many possible ways in which test selection might be performed and optimized. The decisions made at this stage will largely dictate the performance of detection phase.

Very broadly we can categorize the strategies as follows, but there may be significant differences between systems that have been placed within the same category.

## Manual

It is left to the user to select the tests to be run against a mutant. These systems are not designed to be integrated into a build, but instead provide an interface by which a user can select an individual class to mutate and the tests to run.

This approach can only be used to determine the coverage of individual classes.

## Naive

Test selection is automatic, but little or no attempt is made to pick relevant tests for each mutant. Potentially the entire suite, or a large portion of the test suite, is run against each mutant.

This approach glacially slows for anything except the smallest projects.

## Convention based

Test selection is automatic, with tests selected from the suite based on a naming convention or other simple scheme such as annotations. Optimizations may also be implemented to choose an optimal running order for the tests.

This approach is faster than a naive approach. However, it will not give an accurate picture if mutants are covered by tests not picked up by the convention, and it performs badly for mutants that are not exercised by tests

## Coverage based

Test selection is automatic. Tests are selected by first measuring their line, block or instruction coverage. Only those tests that exercise the line, block or instruction that contains the mutant will be run against it. Optimizations may also be implemented to choose an optimal running order for the tests.

This approach is fast: only those tests that could kill the mutant will be run against it. It performs particularly well for mutants that are not exercised by tests - none will be run. It also provides an accurate picture of the coverage of an entire suite. But some overhead required to measure coverage in this approach.

## My Choice

About the tests selection approach, I temporarily chose naïve approach. Naïve approach is automatic and can cover all the tests and test suits. Though it’s slow, my test target is not a huge program and the time is acceptable. Also, naïve is easier to be implemented.

# MUTANT INSERTION

## Naive

Mutants are generated, the class files written to disk, and a new JVM launched with the mutant on the classpath.

This approach should reliably work with any JVM, and mutants will be active during the construction of static state (singletons, static initializers, etc.)

But it’s slow because a new JVM has to be launched for each mutant.

## Mutant Schemata

A single class is generated that contains all mutants, each mutant is then enabled programmatically. Mutant schemata could be used as part of any scheme for mutant insertion, but makes most sense as a variant of a scheme in which class files are written to disk.

This approach is much faster than the plain naive scheme, and should reliably work with any JVM. However, mutants will not be active during the construction of static state.

## Non Delegating Class Loader

Mutants are held in memory and inserted into the JVM by creating a new classloader that does not delegate to its parent when loading the mutant class.

This approach is also faster than naïve scheme, and mutants will be active during the construction of static state and mutants cannot be accidentally released.

The constrains are, breaking the delegation model can lead to classpath issues, particularly with code that uses XML APIs. And this approach also can require large amounts of permgen space.

## Delegating Class Loader

Mutants are held in memory and inserted into the JVM by creating a new classloader which has the boot classloader as a parent.

Delegating class loader is also fster than naïve scheme and fewer classpath problems than non delegating classloader. It’s slower than non delegating classloader as requires more classloading, and requires more permgen space than non delegating loader.

## Debugger Hotswap

With debugger hotswap, mutants are held in memory and the debugger api used to insert them into a running JVM.

Note the expected performance of this approach is unclear. The debugger can degrade the overall performance of a JVM significantly, but this approach does avoid having to launch a new JVM for each mutant.

## Instrumentation API

Mutants are held in memory and inserted into a JVM using the instrumentation API. It’s faster than classloaders and debugger api.

## My Choice

I chose naïve mutant insertion approach in this project. As the target program is not a large project, and I have few tests and test suites, I think the naïve mutant insertion approach is doable and enough in this project.

# MUTANT DETECTION

There are three approaches to detect mutant. In the naïve approach, test classes are run until all selected tests are run.

In the coarse early exit approach, the selected test classes are run until one of them kills a mutan. This is potentially much faster than the naïve approach, but al tests within a class are run to completion so slower than a finer grained approach.

The third approach is fine early exit, test classes are split into individual test cases which are then run until one of them kills a mutant. This is the most fast approach, but some overhead required to split the test cases. And splitting tests out of classes may cause issues with some JUnit extensions.

# MUTATORS

Mutations are performed on the byte code generated by the compiler rather than on the source files3. This approach has the advantage of being generally much faster and easier to incorporate into a build, but it can sometimes be difficult to simply describe how the mutation operators map to equivalent changes to a Java source file.

The mutators I introduced in this section, are what I am going to use to in this project.

## Conditionals Boundary Mutator

The conditionals boundary mutator replaces the relational operators <, <=, >, >=.

with their boundary counterpart as per the table below.

Table 1. Conditionals Boundary Mutation

|  |  |
| --- | --- |
| **Original conditional** | **Mutated conditional** |
| < | <= |
| <= | < |
| > | >= |
| >= | > |

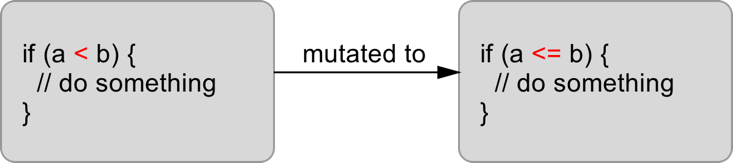


Figure . Conditionals Boundary Mutator Example

## Negate Conditionals Mutator

The negate conditionals mutator will mutate all conditionals found according to the replacement table below.

**Table 2. Negate Conditionals Mutation**

|  |  |
| --- | --- |
| **Original conditional** | **Mutated conditional** |
| ＝＝ | != |
| != | == |
| <= | > |
| >= | > |
| < | >= |
| > | <= |

## Remove Conditionals Mutator

The remove conditionals mutator will remove all conditionals statements such that the guarded statements always execute.

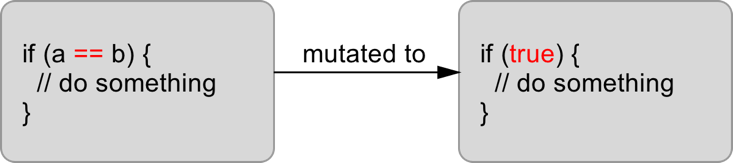


Figure . Remove Conditionals Mutation Example

## Math Mutator

The math mutator replaces binary arithmetic operations for either integer or floating-point arithmetic with another operation. The replacements will be selected according to the table below.

**Table 3. Math Mutation**

|  |  |
| --- | --- |
| **Original conditional** | **Mutated conditional** |
| + | - |
| - | + |
| \* | / |
| / | \* |
| % | \* |
| & | | |
| | | & |
| ^ | & |
| << | >> |
| >> | << |
| >>> | << |

## Increments Mutator

The increments mutator will mutate increments, decrements and assignment increments and decrements of local variables (stack variables). It will replace increments with decrements and vice versa.

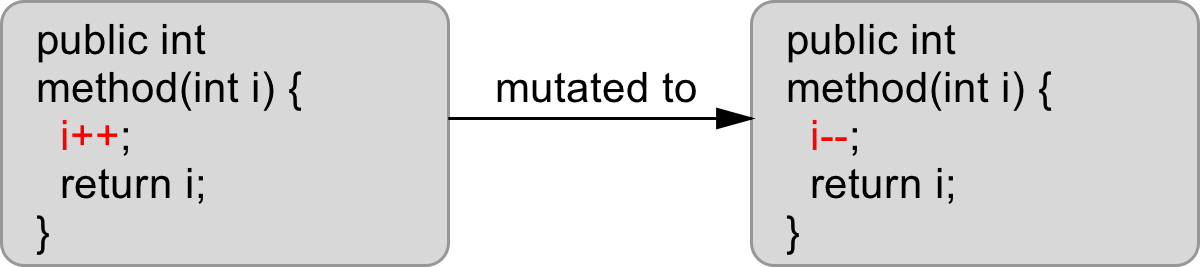


Figure . Increments Muator Example

## Invert Negatives Mutator

The invert negatives mutator inverts negation of integer and floating point numbers.

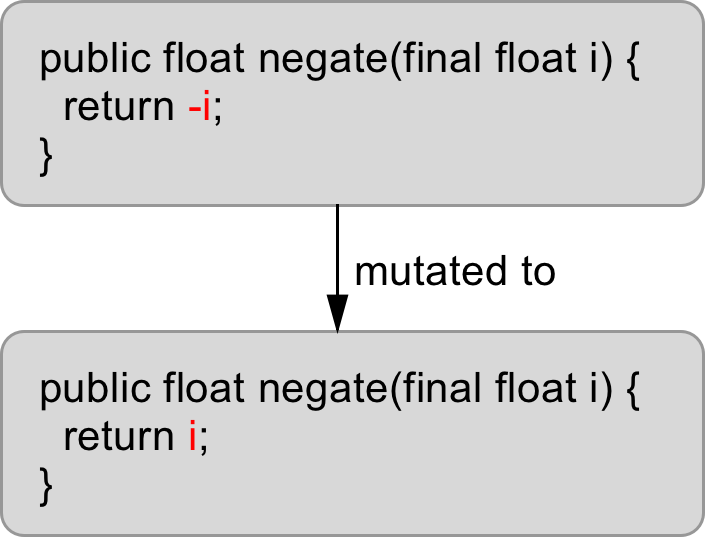


Figure . Invert Negatives Mutator Example

## Return Values Mutator

The return values mutator mutates the return values of method calls. Depending on the return type of the method another mutation is used.

**Table 3. Return Values Mutation**

|  |  |
| --- | --- |
| **Return Type** | **Mutation** |
| boolean | Replace the unmutated return value true with false and replace the unmutated return value false with true. |
| int  byte  short | If the unmutated return value is 0 return 1, otherwise mutate to return value 0. |
| long | Replace the unmutaed return value x with the result of x+1. |
| float  double | Replace the unmutated return value x with the result of –(x+1.0) if x is not NAN and replace NAN with 0. |
| Object | Replace non-null return values with null and throw a java.lang.RuntimeException if the unmutated method would return null. |

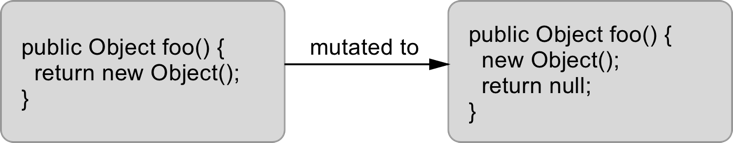


Figure . Return Values Mutator Example

## Void Method Call Mutator

The void method call mutator removes method calls to void methods.

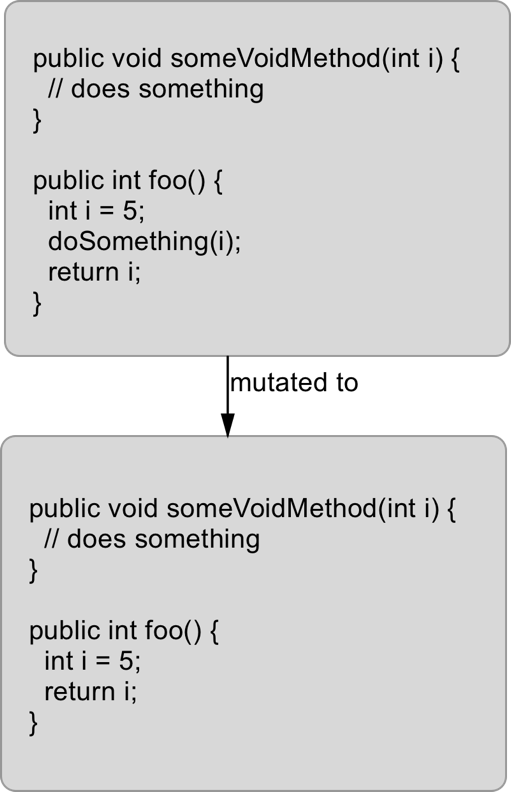


Figure . Void Method Call Mutator Example

# ACKNOWLEDGMENTS

Our thanks to ACM SIGCHI for allowing us to modify templates they had developed.

# REFERENCES

1. *Mutation testing systems for Java compared.* (2016, March). Retrieved  from http://pitest.org/java\_mutation\_testing\_systems/
2. *ASM*. (2016, March). Retrieved  from http://asm.ow2.org
3. *Mutation operators.* (2016, March). Retrieved  from http://pitest.org/quickstart/mutators/