**Final Project Report of**

**Software Testing, Validation and Verification**

|  |  |
| --- | --- |
| Ning Xin  nxx170130@utdallas.edu  Github: <https://github.com/dumblejd/CS6367> | Di Jin  dxj170930@utdallas.edu |

**1 PROBLEM INTRODUCTION**

PIT currently provides some built-in mutators, of which most are activated by default. The default set can be overridden, and different operators selected, by passing the names of the required operators to the mutators parameter.

Mutations are performed on the byte code generated by the compiler rather than on the source files. 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 operators are largely designed to be stable (i.e not be too easy to detect) and minimise the number of equivalent mutations that they generate. Those operators that do not meet these requirements are not enabled by default.

The problem is to contribute to the PIT mutation testing tool to implement more mutation operators. The original set of mutation operators implemented in PIT does not include some traditional mutation operators.

The problem is divided into 2 parts. The first part is to augment the PIT tool to implement the following mutation operators, and evaluate the augmented PIT tool on 5 real-world projects from GitHub(with > 1000 lines of code and > 50 tests):

1. AOD: Replaces an arithmetic expression by each one of the operand, e.g., a + b ➔ a and a + b ➔ b.
2. ROR: Replaces the relational operators with each of the other ones. It applies every replacement, e.g., < ➔ >=, < ➔ !=, and < ➔ <=.
3. AOR: Replaces an arithmetic expression by each of the other ones, e.g., a + b ➔ a \* b

The second part is to further augment the PIT mutation testing tool with more code-fixing rule to automatically fix real-world program bugs from Defects4J. We are going to add the following mutators in this part:

1. M1: For each field dereference, such as o.f, add a conditional checker to perform the field dereference only when o!=null.
2. M2: Replaces a method invocation with a call to another overloading method (i.e., method with the same name but different arguments). When the new overloading method has less arguments than the original method, simply reuse a subset of the prior arguments; when the new overloading method has more arguments, use null or some default value [8] as the additional argument values. Note that the new call should have the same return type as the old call.
3. M3: Replaces a method invocation with a call to another method with the same method descriptor (e.g., return type and argument types). The new callee should have a different name, but its return type and argument types should be compatible with what is expected of the calling context.
4. M4: In each expression e, the operator replaces each variable occurring in the expression by another variable with a compatible type.

**2 TECHNIQUES DETAILS**

**2.1 ASM bytecode engineering framework**

ASM is an all purpose Java bytecode manipulation and analysis framework. It can be used to modify existing classes or dynamically generate classes, directly in binary form. Provided common transformations and analysis algorithms allow to easily assemble custom complex transformations and code analysis tools.

ASM offer similar functionality as other bytecode frameworks, but it is focused on simplicity of use and performance. Because it was designed and implemented to be as small and as fast as possible, it makes it very attractive for using in dynamic systems\*.

**2.2 JavaAgent**

Provides services that allow Java programming language agents to instrument programs running on the JVM. The mechanism for instrumentation is modification of the byte-codes of methods.

**2.3 Maven Build System**

Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.

Maven’s primary goal is to allow a developer to comprehend the complete state of a development effort in the shortest period of time. In order to attain this goal there are several areas of concern that Maven attempts to deal with:

Making the build process easy

Providing a uniform build system

Providing quality project information

Providing guidelines for best practices development

Allowing transparent migration to new features

**3 IMPLEMENTATION PLAN**

Phase one:

Prerequisite Knowledge 2/7-2/14

Official document understanding 2/14-2/16

Analyse original PITest code 2/16-2/23

Create own mutators and test on real-world code 2/23-3/15

Conclusion and experiment analyse 3/15-3/20

Phase two:

Improve mutators in phase one 3/22-3/26

Analyse circumstance for mutator bug-fixing 3/27-4/4

Implement more mutators 4/4-4/10

Try bug-fixing on real-word code 4/11-4/15

Modify and analyse mutators combination 4/16-4/18

Conclusion and experiment analyse 4/18-4/20

**4 EXPERIMENTAL EVALUEATION AND DESIGN**

Phase one:

Understand the ASM bytecode engineering framework, the JavaAgent on-the-fly code instrumentation, and the Maven build system.

Check the official document of PIT for basic available mutators.

Understand the original mutators in PIT on Github.

Create AOD, ROR, AOR mutators instance:

Create new instances of MethodMutatorFactory.

Create new MethodVistor to do the work of creating the mutant.

Modify org.pitest.mutationtest.engine.gregor.config.Mutator to inform the PIT the new mutators.

Try customized mutators on real-world projects from Github.

Phase two:

Modify mutators for correctness if needed and update them on Github.

Analyse possibility of fixing bug in different conditions with mutators.

Try to implement more mutators to realize code-fixing rules.

Evaluate the fixing of the real-world program.

Create more customized mutators and try to fix the same program.

Compare the bug-fixing ability of different amount of mutators.

The experiment is evaluated according the completeness of the customized mutators and effort on the bug-fixing by use different mutators.

**5 PROGRESS**

**PREPARATION**

Before staring creating the customized mutator, we read the official pitest code on GitHub. Also, the author gives three step to create mutator: Firstly, create a new instance of MethodMutaorFactory. Then, create a new MethodVisitor to do the work of creating the mutant. Finally, update configuration of pitest to inform the mutator. Step two is the hardest part since the change of the code is all about byte code which differs a lot from normal Java code. E.g. Arithmetic expression is processed in stack with rule of first in last out.

By applying the mutation operators PIT will generate a number (potentially a very large number) of mutants. These are Java classes which contain a mutation (or fault) which should make them behave differently from the unmutated class.

PIT will then run tests using this mutant instead of the umutated class. An effective set of tests should fail in the presence of the mutant.

Then we clone the pitest from GitHub to our repository. Because the GitHub allows multiple developer to work together on one project. Firstly we create a small maven project to test the original pitest mutators.

PIT runs unit tests against the mutated code automatically. Before running the tests PIT performs a traditional line coverage analysis for the tests, then uses this data along with the timings of the tests to pick a set of test cases targeted at the mutated code.

This approach makes PIT much faster than previous mutation testing systems such as Jester and Jumble, and enables PIT to test entire code bases, rather than single classes at a time.

By using pom.xml to inform maven about the environment for the testing including junit and pitest. In commend line, we use pitest-maven:mutationCoverage to run the test. Which gives a report about the line coverage and mutator coverage. According to the report, the original pitest can run successfully in our environment.

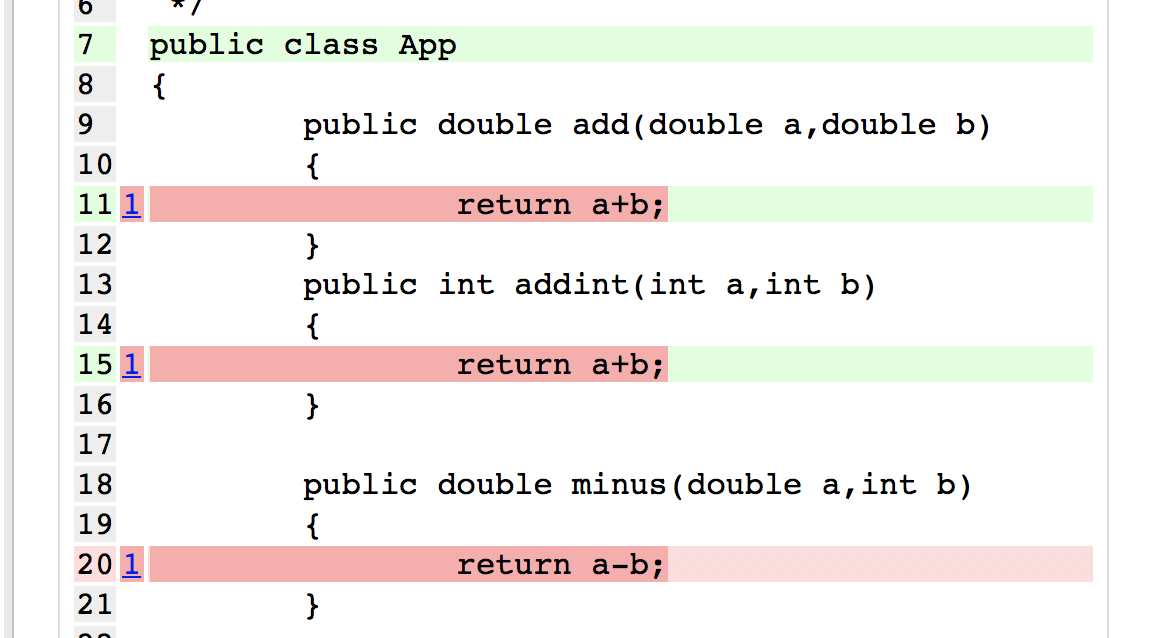
**CREATE MUTATOR**

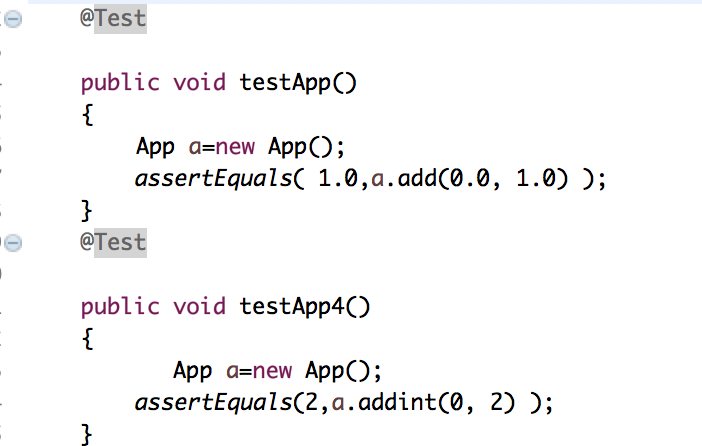
AOD: Replaces an arithmetic expression by each of the operand, e.g., a + b -> a

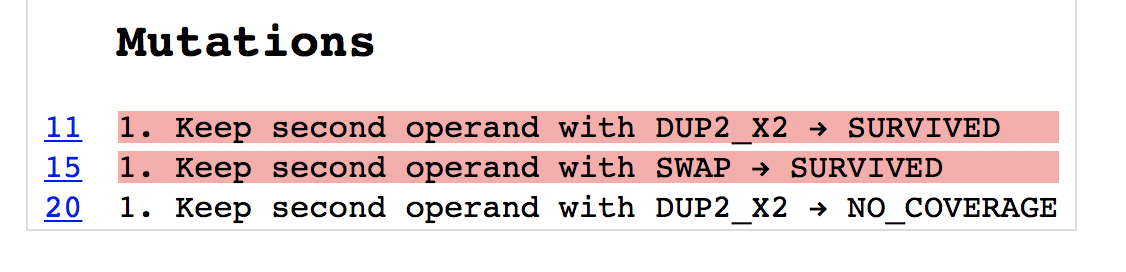
ROR: Replaces the relational operators with each one. It applies every replacement, e.g., < to ≥, or > to ≤

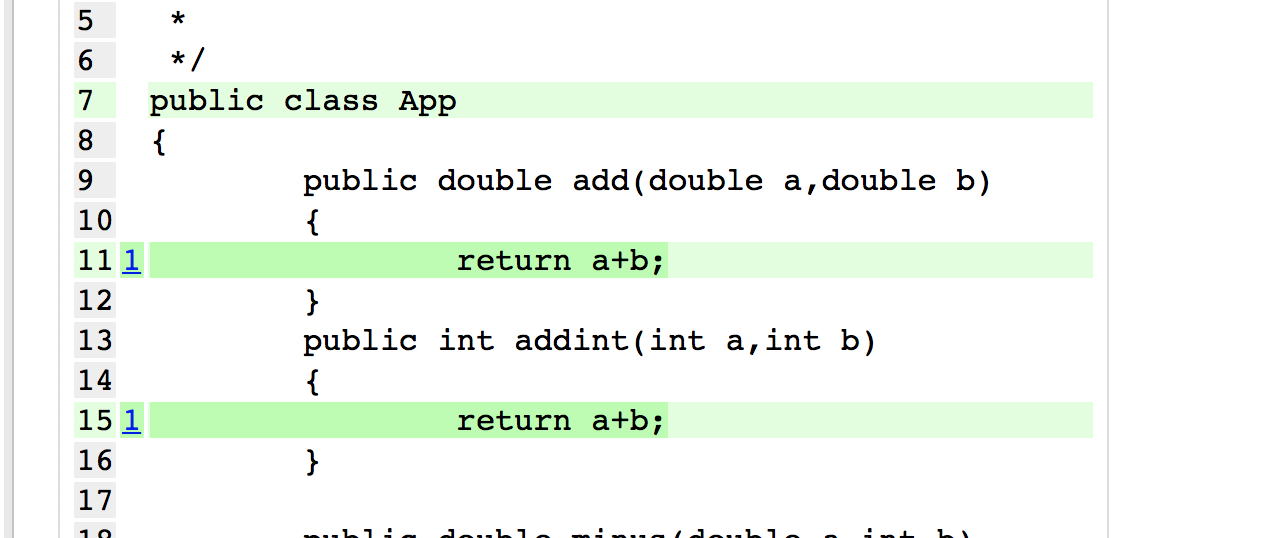
AOR: Replaces an arithmetic expression by each one. a + b -> a ∗ b

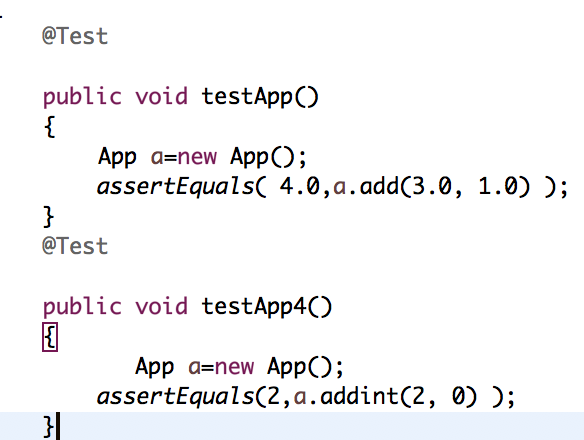
We use Opcode.POP and Opcode.POP2 to implement the AOD function according to the logic of stack. It will pop the top character of the stack by which the first character is remained. To test this function, we use simple plus function. For 1+2=3 should be killed but 1+0=1 will survive since after popping 0, 1 still equal to 1.

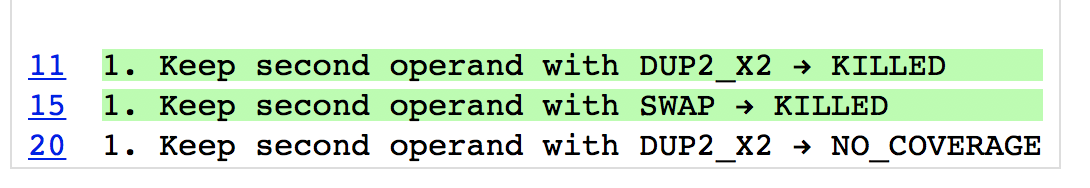




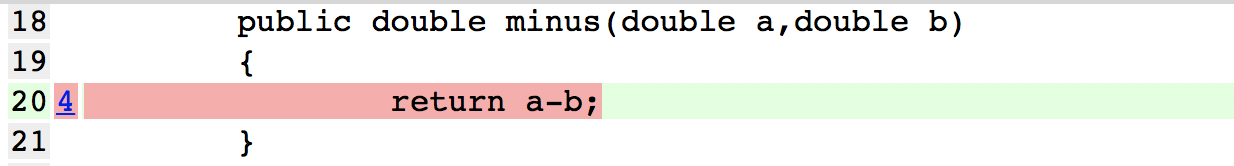


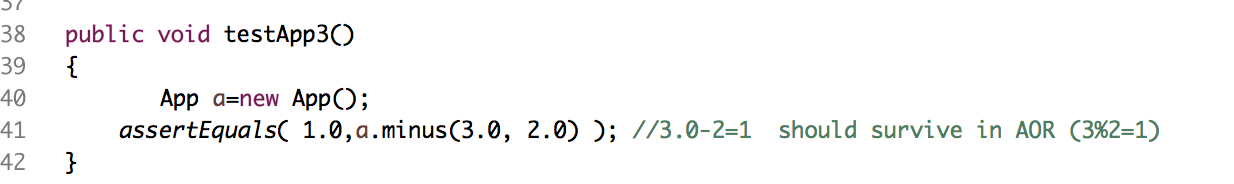


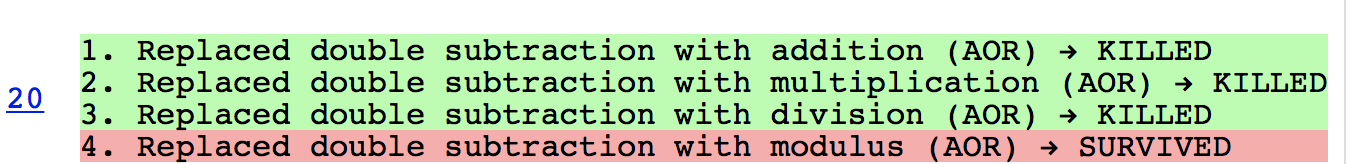


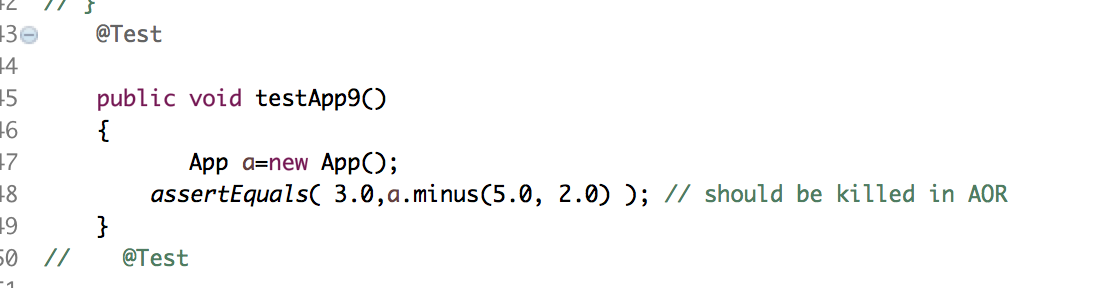


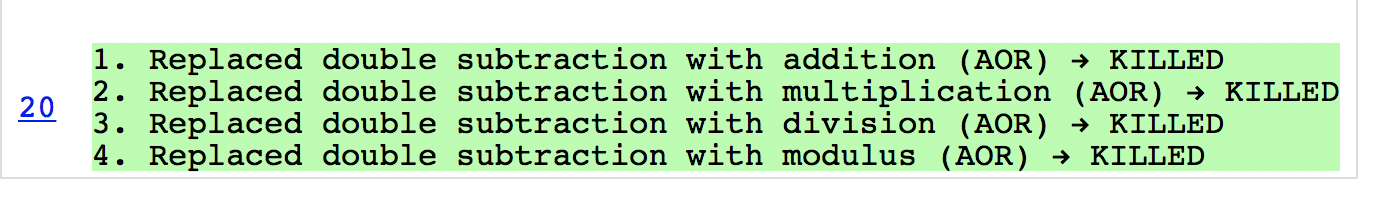
For AOR, we use the same trick by creating a different Opcode operator. For 5-2=3 should be killed since for all math operator that replace ‘-’ will not make the result equal to 3. While 3-2=1 will survive since 3%2=1.



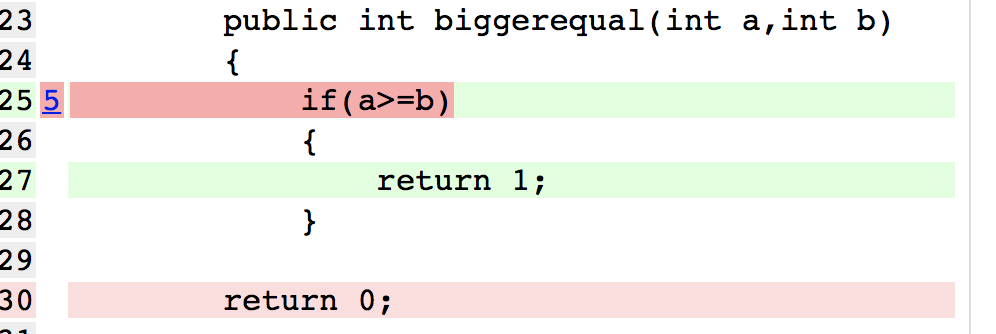


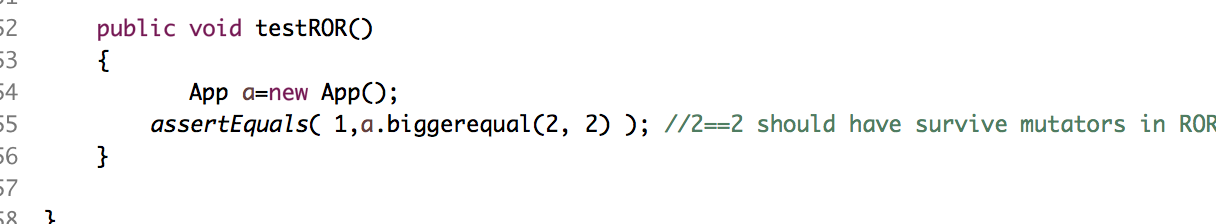


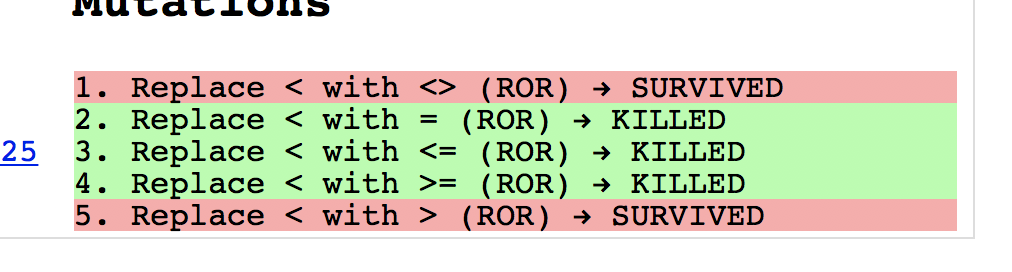




For ROR we replace relational operators with each one. For the condition 2>=2 will return false in replacement of >=. Some similar conditions are tested for the correctness of the mutator.



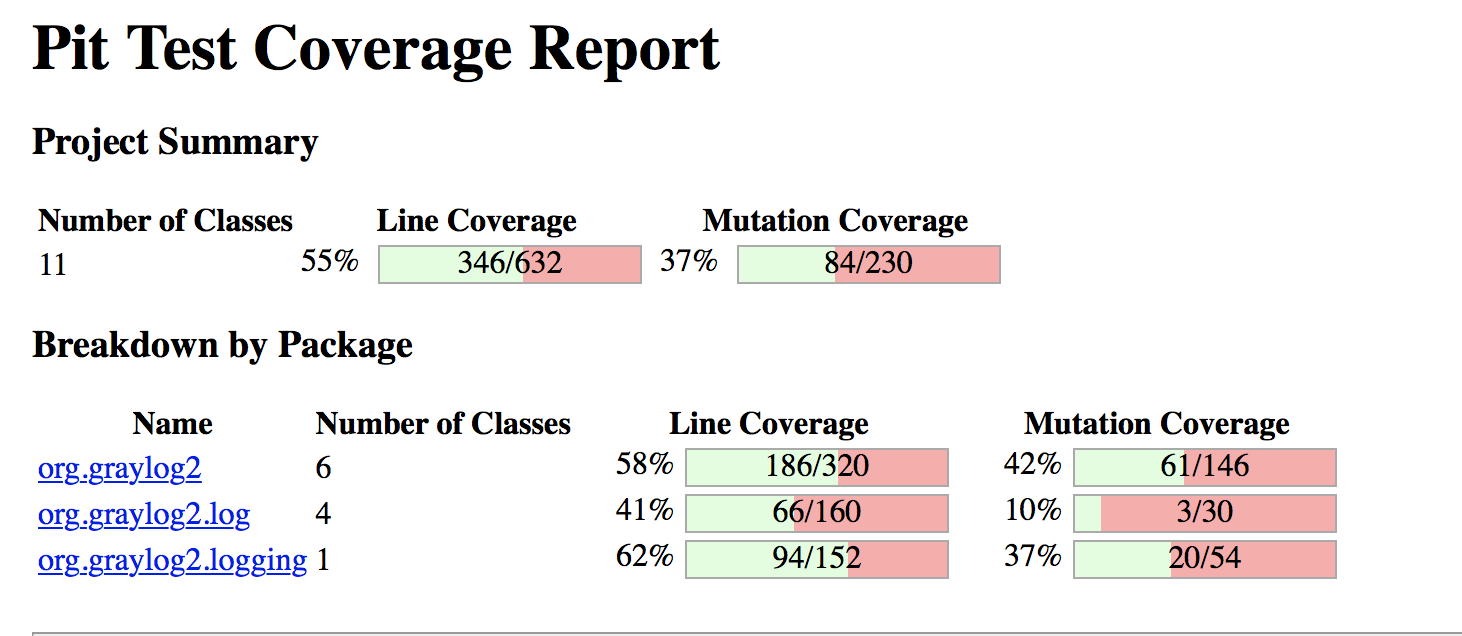




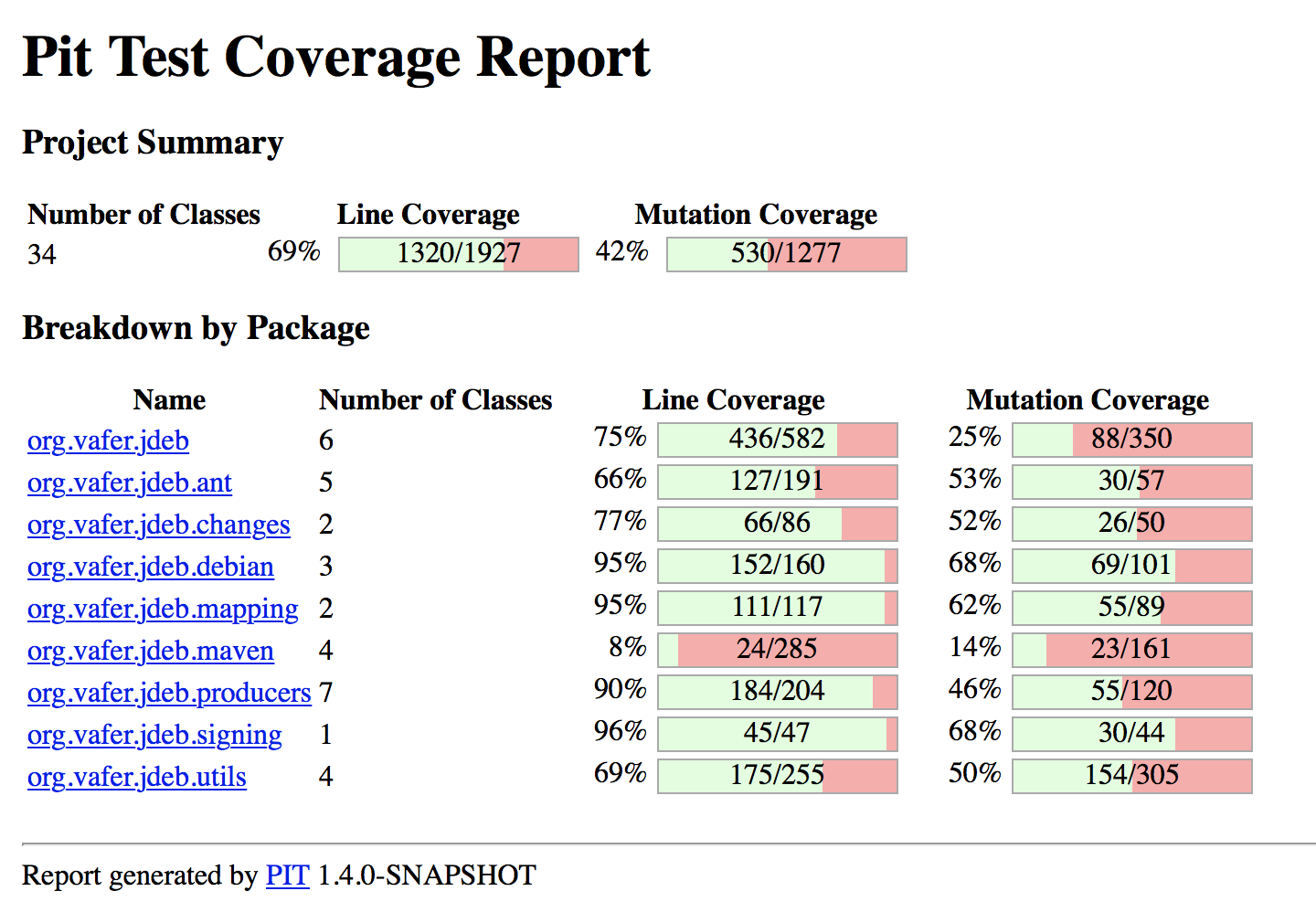
**TESTING MUTATOR ON REAL WORLD PROJECT**

We select five projects on GitHub with test that have already been written.

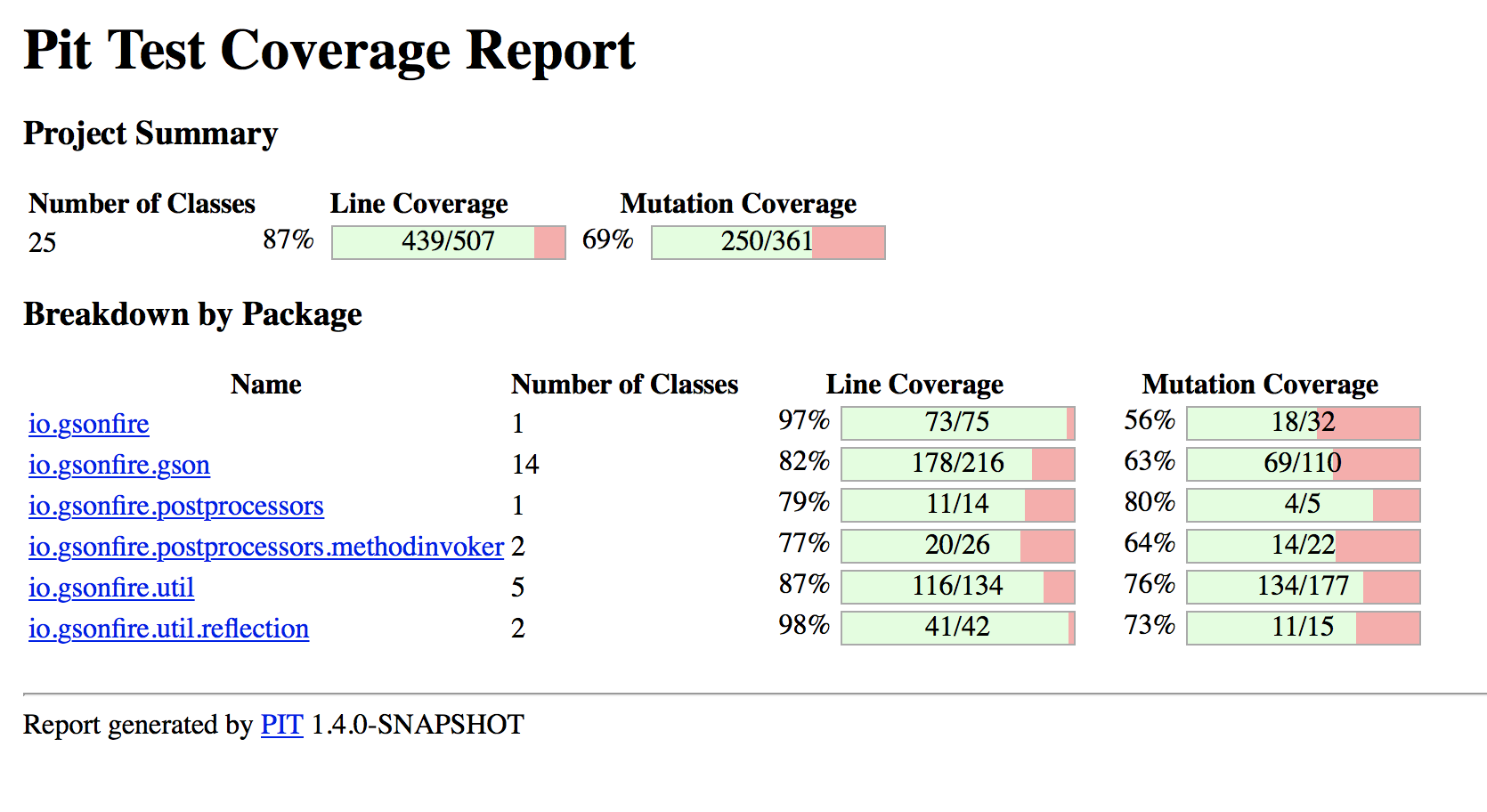
https://github.com/t0xa/gelfj.git



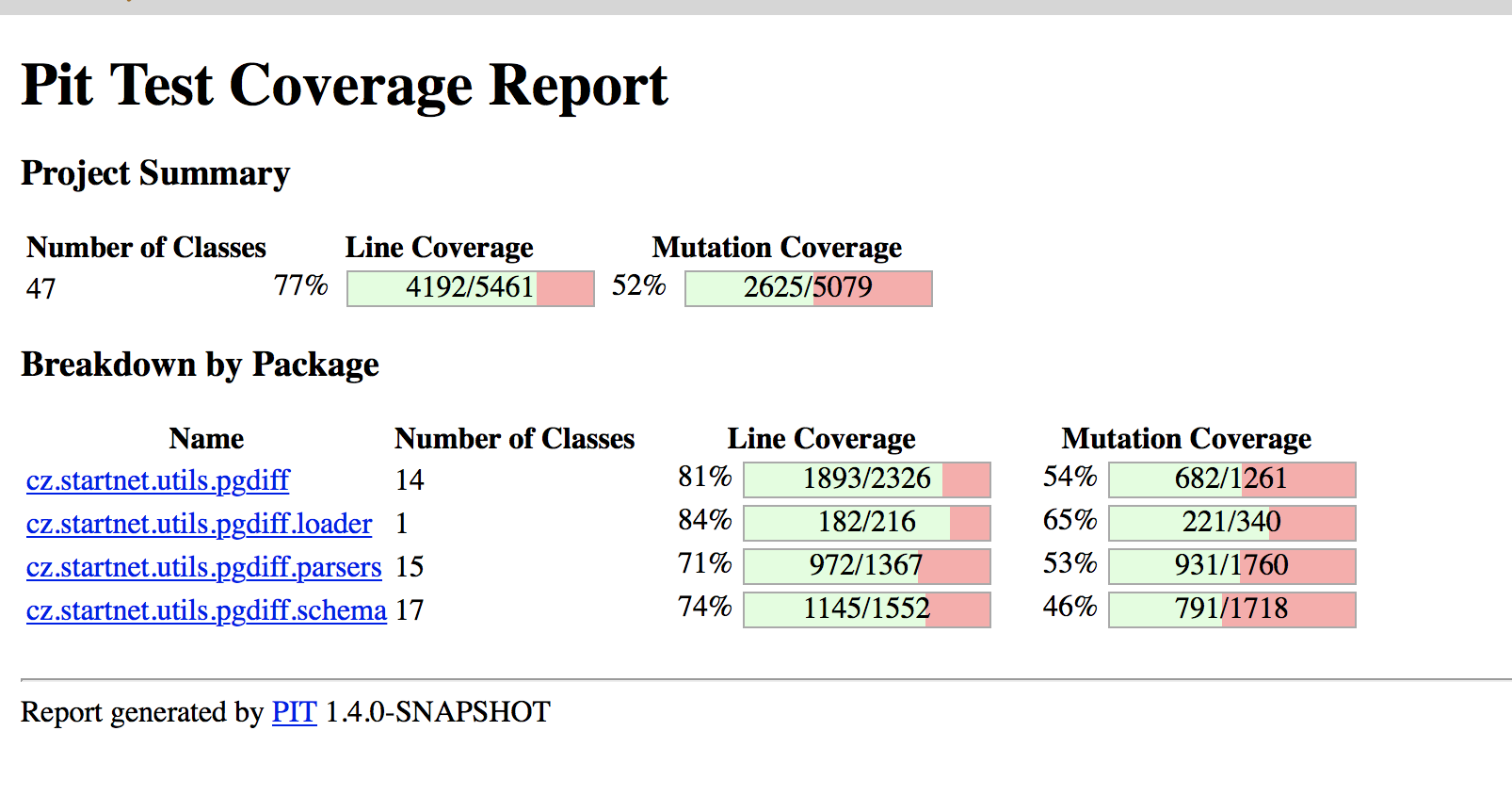
<https://github.com/tcurdt/jdeb.git>



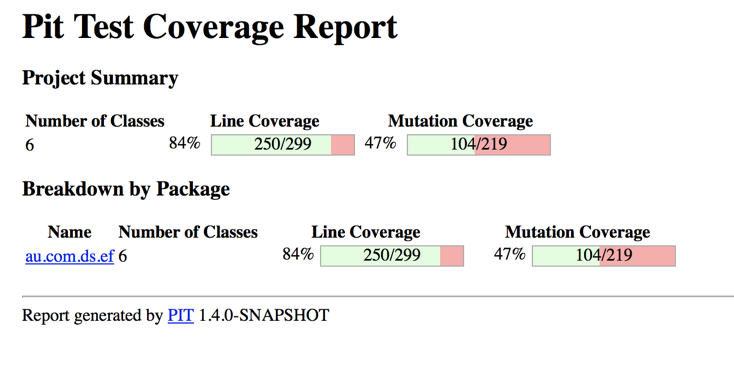
https://github.com/julman99/gson-fire.git



<https://github.com/fordfrog/apgdiff.git>



<https://github.com/Beh01der/EasyFlow.git>



**6 Further Implemented M1 and M4**

The second part is to further augment the PIT mutation testing tool with more code-fixing rule to automatically fix real-world program bugs from Defects4J. We are going to add the following mutators in this part:

1. M1: For each field dereference, such as add a conditional checker to perform the field dereference only when o!=null.
2. M4: In each expression e, the operator replaces each variable occurring in the expression by another variable with a compatible type.

In M4, we use a list of String to store all types of variables, and a list of Integer to store all indexes of variables in byte code level. When the program loading the variable(int, long, float, double, object), we check if the list contains the index of that variable. If it does, we get the index of that variable in the list, and generate the replacement variable of the same type. When the program storing the variable(int, long, float, double, object), we check the list contains the index of this variable, if not, we add the index of it to the list. Also, we check the type of this variable, and the type of it add to the list.

For example,

int a = 0,

int c = 0,

a = 3;

return a;

By using M4, the result should be 0.