Software engineering project report
Mutation Testing

戴凡淞 1909853L-I011-0021

孙宸 1909853T-I011-0024

康宇为 1909853J-I011-0051

易林蔚 1909853M-I011-0074

Abstract

Mutation testing (sometimes called "mutation analysis") is a software testing method that can improve the source code of a program in detail. These so-called mutations are based on well-defined mutation operations. In this project, we used Milu to mutate a project Calculator-master downloaded from GitHub, which is a scientific calculator. We made some changes to the code of this project to automatically read test cases, and then selected 20 A mutant, and write code to collect the results of each mutant, and compare with the real results to get a mutation matrix. Through the analysis of the mutation matrix, we found that our test cases can cover 100% of a purpose function. We also uploaded our code to GitHub. The link is as follows: https://github.com/MeditatorE/Mutation- testing

1. Introduction

Developers always produce certain errors when developing software daily. In order to find these errors as soon as possible and correct them as soon as possible, in the process of software development, developers often need to test the software, so the developer needs to write for the test. Or generate some test cases. But how can we confirm whether the test case used can cover 100% of the code? Mutation testing is a commonly used method. Mutation Testing (sometimes called "mutation analysis") is a software testing method that improves the source code of a program in detail. These so-called mutations are based on well-defined mutation operations. [3] These operations either simulate typical application errors (for example: using wrong operators or variable names), or force effective tests (for example, making every expression equal to 0). The purpose is to help testers find effective testing, or locate weaknesses in test data, or weaknesses in code that are rarely (or never) used in execution.

2. Original Project details

To implement Mutation Testing, our group selected and downloaded a project called Calculator-master [2] from GitHub. This project realizes part of the functions of scientific calculators, including 20 common function operations and parenthesis nesting operations. The specific functions are as follows: SIN, COS, TAN, ARCSIN, ARCCOS, ARCTAN, SINH, COSH, TANH, LOG10, LN, EXP, FACT, SQRT, CUBEROOT, LOG, POW, MOD, YROOT, AVG, SUM, VAR, VARP, STDEV, STDEVP. The idea of this project is to detect function names through semantics, divide expressions, and encapsulate them with data structures. Two stacks, one is used to represent the nested structure between functions, and the other is used to calculate operation expressions. Our group believes that this project can be implemented well and clearly demonstrate the Mutation Testing process.

3. Method

Since there are fewer mutation tools in the C language, our group finally chose the Milu [1] mutation tool to mutate the project. We mutated other tool functions in the project except for the main function (Figure 1). Since mutations are random, there may be one mutation or multiple mutations in a tool function. We don't know the specific location of the mutation, so we manually check the mutant, find out the mutation location and mark it. Figure 1 shows an example of a mutation. A total of 40 versions of mutations like this were generated, from which we randomly selected 20 versions as our research objects and wrote test cases (Figure 9) for testing.

```
long double ARCTAN(int number,long double count[],int len)
{
    if(function_array[number].std_number==len)
    {
            return atan(count[0]);
    }
    return SINT;
}
```

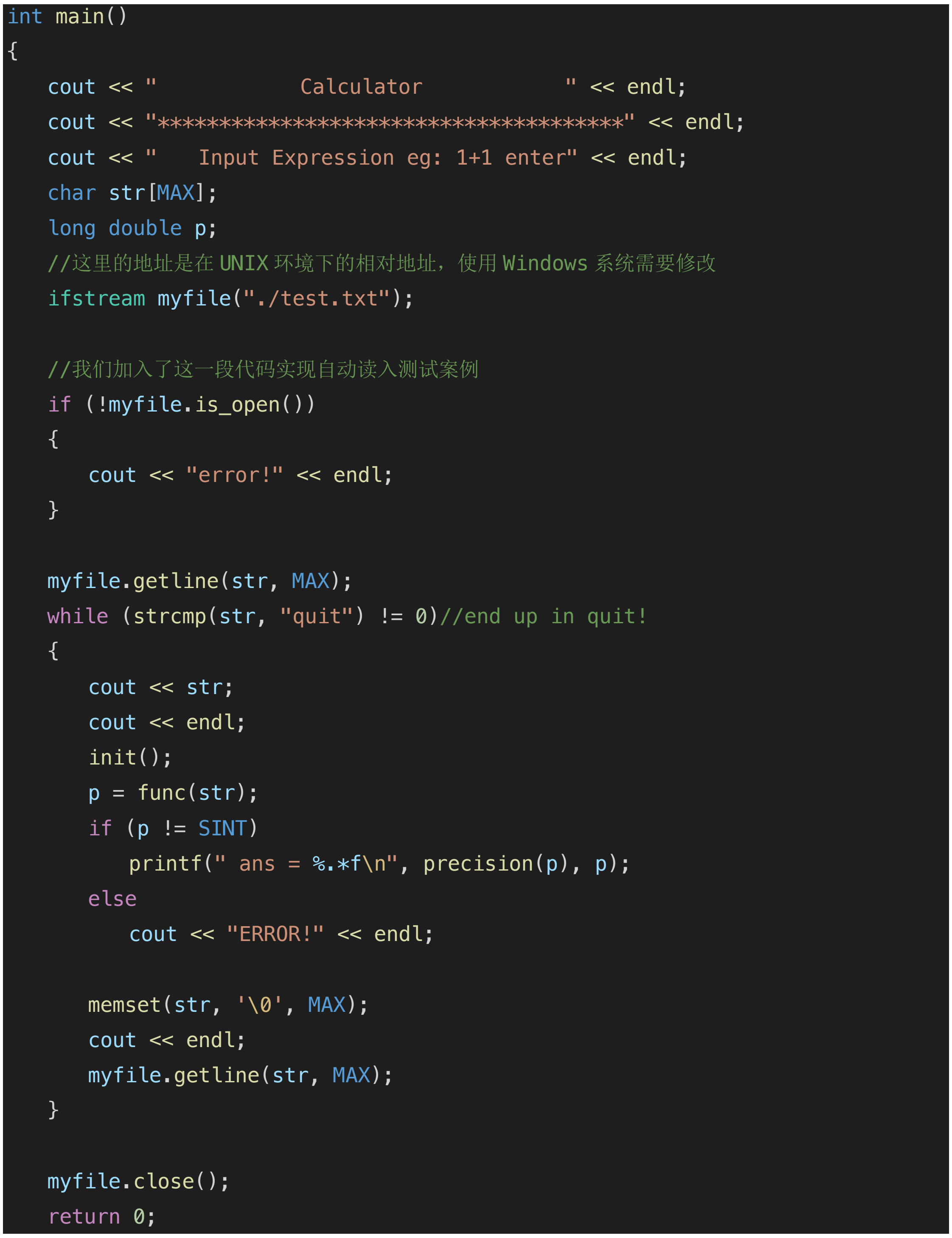
Original code

```
long double ARCTAN(int number,long double count[],int len)
{

if(function_array[number].std_number!=len)//if(function_array[number].std_number==len)
    {
            return atan(count[0]);
    }
    return SINT;
}
```

Mutant

Figure 1. Mutant example

In order to improve efficiency and automatically perform mutation testing, we have made some modifications to the code of the original project. The code shown in Figure 2 is added to the code file. The function of this part of the code is to help the target project automatically read the test case we wrote. In addition, a script file (Figure 3) can help us automatically complete the mutation test, and automatically save the output result to a file named result.txt (Figure 6). The purpose of generating this txt file is to generate a mutation matrix.



```
int main()
{
    cout << "            Calculator            " << endl;
    cout << "***************************************" << endl;
    cout << "   Input Expression eg: 1+1 enter" << endl;
    char str[MAX];
    long double p;
    //这里的地址是在 UNIX 环境下的相对地址，使用 Windows 系统需要修改
    ifstream myfile("./test.txt");

    //我们加入了这一段代码实现自动读入测试案例
    if (!myfile.is_open())
    {
        cout << "error!" << endl;
    }

    myfile.getline(str, MAX);
    while (strcmp(str, "quit") != 0)//end up in quit!
    {
        cout << str;
        cout << endl;
        init();
        p = func(str);
        if (p != SINT)
            printf(" ans = %.*f\n", precision(p), p);
        else
            cout << "ERROR!" << endl;

        memset(str, '\0', MAX);
        cout << endl;
        myfile.getline(str, MAX);
    }

    myfile.close();
    return 0;
}
```

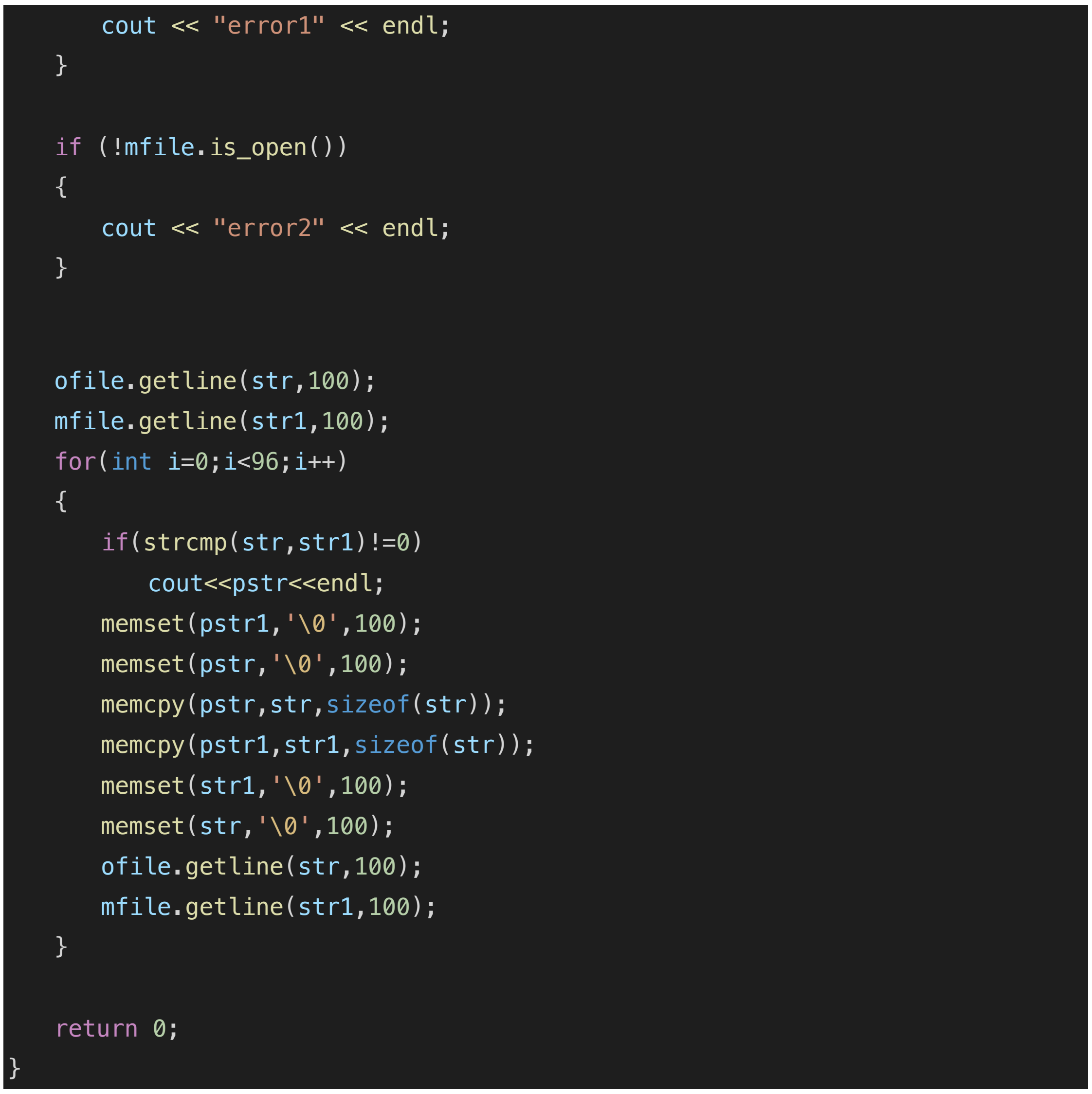
Figure 2. Modified code

```
#这个脚本文件用于自动运行代码并将结果重定向到 result.txt 中
cd ./Calculater/ #基于 UNIX 的地址 Windows 需要修改
g++ Calculater.cpp
./a.out >> result.txt
mv result.txt key.txt

for ((i=0; i<20; i++))
do
    cd ~
    cd ./mutation/$i/
    g++ Calculater.cpp
    ./a.out >> result.txt
    mv result.txt $i.txt
done
```

Figure 3. Script code

In order to obtain the mutation matrix more simply, we have written a code (Figure 4) whose function is to compare the result.txt file generated by each mutant in the previous step with the correct result generated by the original project, and compare the differences between the two results. The place is automatically recorded in another matrix.txt file (Figure 9) to generate a mutation matrix.

```
#include<cstdio>
#include<iostream>
using namespace std;
#include<cstdlib>
#include <vector>
#include <string>
#include <fstream>

int main()
{
    char str[100];
    char str1[100];
    char pstr[100];
    char pstr1[100];
    ifstream ofile("./origral.txt"); //项目的原始结果,需要改名
    ifstream mfile("./result.txt");  //突变体产生的结果，需要改名

    if (!ofile.is_open())
    {
```

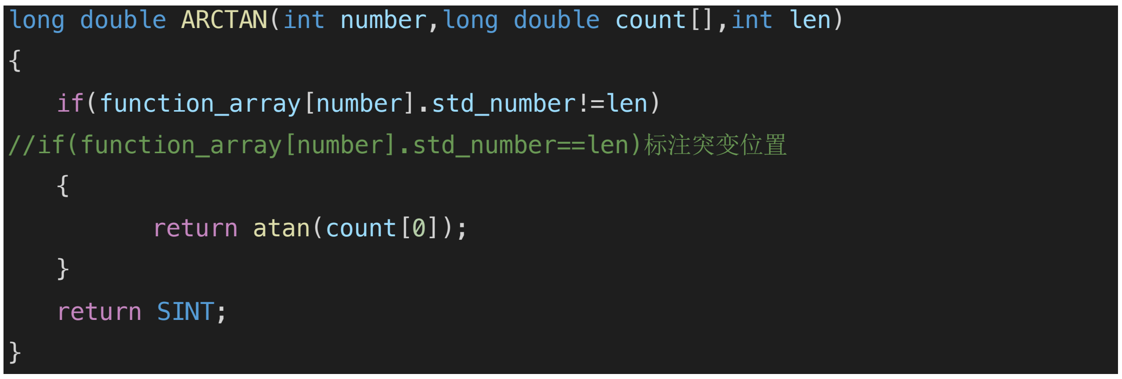


```
        cout << "error1" << endl;
    }

    if (!mfile.is_open())
    {
        cout << "error2" << endl;
    }


    ofile.getline(str,100);
    mfile.getline(str1,100);
    for(int i=0;i<96;i++)
    {
        if(strcmp(str,str1)!=0)
            cout<<pstr<<endl;
        memset(pstr1,'\0',100);
        memset(pstr,'\0',100);
        memcpy(pstr,str,sizeof(str));
        memcpy(pstr1,str1,sizeof(str));
        memset(str1,'\0',100);
        memset(str,'\0',100);
        ofile.getline(str,100);
        mfile.getline(str1,100);
    }

    return 0;
}
```

Figure 4. Mutation matrix generates code

4. Experiment

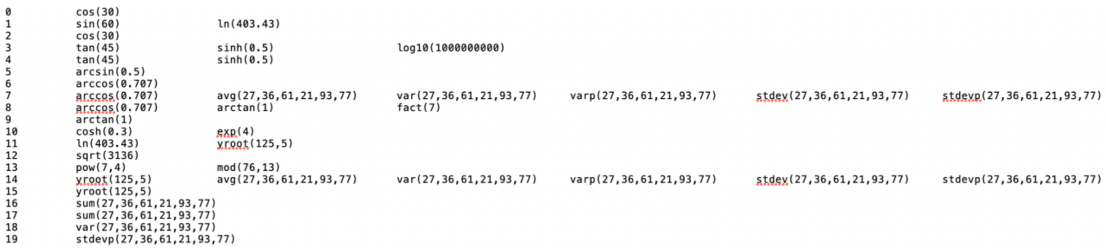
We analyzed the generated mutation matrix. In order to more comprehensively evaluate whether all mutation cases were found in our test case, we manually checked the location of mutation in each mutant and matched it with the test case (Figure 7).



```
long double ARCTAN(int number,long double count[],int len)
{
    if(function_array[number].std_number!=len)
//if(function_array[number].std_number==len)标注突变位置
    {
            return atan(count[0]);
    }
    return SINT;
}
```

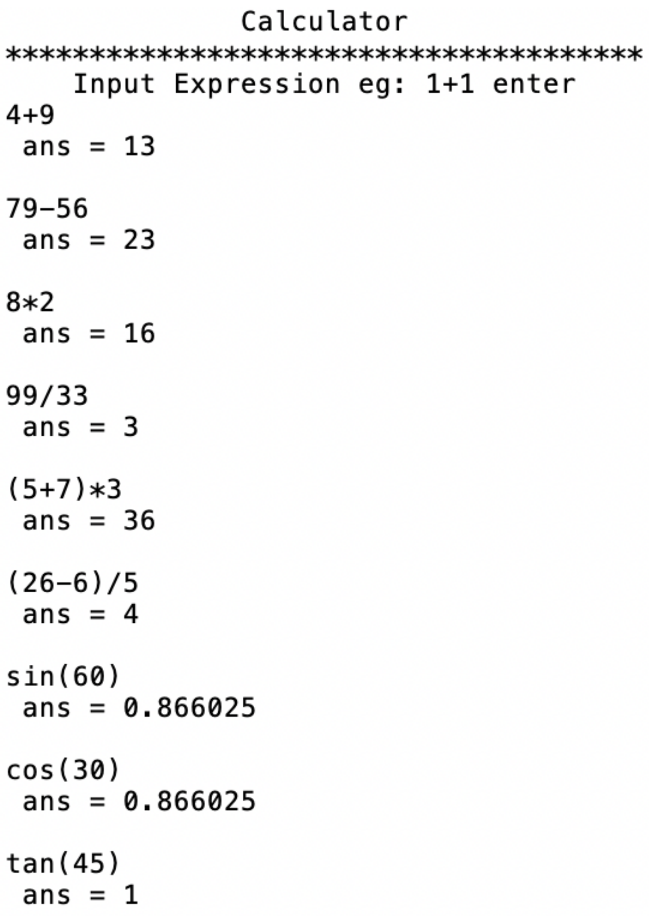
Figure 7. Mark the location of the mutation

Finally we get the mutation matrix (Figure 8) and analyze it， We found that the test cases we used did produce anomalies at each mutation position, and through the test cases that generated anomalies, we could also infer the function of collectively generating mutations.

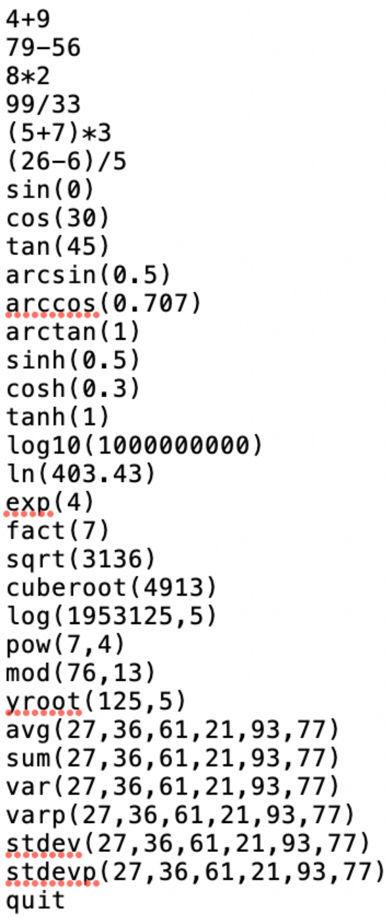


0	cos(30)					
1	sin(60)	ln(403.43)				
2	cos(30)					
3	tan(45)	sinh(0.5)	log10(1000000000)			
4	tan(45)	sinh(0.5)				
5	arcsin(0.5)					
6	arccos(0.707)					
7	arccos(0.707)	avg(27,36,61,21,93,77)	var(27,36,61,21,93,77)	varp(27,36,61,21,93,77)	stdev(27,36,61,21,93,77)	stdevp(27,36,61,21,93,77)
8	arccos(0.707)	arctan(1)	fact(7)			
9	arctan(1)					
10	cosh(0.3)	exp(4)				
11	ln(403.43)	yroot(125,5)				
12	sqrt(3136)					
13	pow(7,4)	mod(76,13)				
14	yroot(125,5)	avg(27,36,61,21,93,77)	var(27,36,61,21,93,77)	varp(27,36,61,21,93,77)	stdev(27,36,61,21,93,77)	stdevp(27,36,61,21,93,77)
15	yroot(125,5)					
16	sum(27,36,61,21,93,77)					
17	sum(27,36,61,21,93,77)					
18	var(27,36,61,21,93,77)					
19	stdevp(27,36,61,21,93,77)					

Figure 8. **Mutation matrix.** The number in the first column is the serial number of the mutant, and the following formula is the test case



```
                 Calculator
***************************************
    Input Expression eg: 1+1 enter
4+9
 ans = 13

79-56
 ans = 23

8*2
 ans = 16

99/33
 ans = 3

(5+7)*3
 ans = 36

(26-6)/5
 ans = 4

sin(60)
 ans = 0.866025

cos(30)
 ans = 0.866025

tan(45)
 ans = 1
```



```
4+9
79-56
8*2
99/33
(5+7)*3
(26-6)/5
sin(0)
cos(30)
tan(45)
arcsin(0.5)
arccos(0.707)
arctan(1)
sinh(0.5)
cosh(0.3)
tanh(1)
log10(1000000000)
ln(403.43)
exp(4)
fact(7)
sqrt(3136)
cuberoot(4913)
log(1953125,5)
pow(7,4)
mod(76,13)
yroot(125,5)
avg(27,36,61,21,93,77)
sum(27,36,61,21,93,77)
var(27,36,61,21,93,77)
varp(27,36,61,21,93,77)
stdev(27,36,61,21,93,77)
stdevp(27,36,61,21,93,77)
quit
```

Figure 6. result.txt

Figure 9. Test case

5. Conclusion

By analyzing the mutation matrix in Figure 9, we found that the test cases we used in Figure 6 can completely cover all the functional functions in this project, which means that our test cases cover the functional functions in this project. Can reach 100%. But our experiment may have a shortcoming. For other non-functional functions in this project, our test cases may not be able to accurately detect the location of mutations. Even if the mutation of non-functional functions does not affect the normal operation of this project, our test cases may not recognize this mutant. Fortunately, such mutations have almost no effect on the results of this project, so we can draw a conclusion that our test cases can perfectly kill all mutants that may have a negative impact on the project.

6. Instructions

There are four people in our team and the specific division of labor is as follows:

Sun Chen
Write the testing code, Write the report, Analyze the result data

Dai Fansong
Write testing cases, Conduct mutation testing, Test original projects, and Proofread data

Yi Linwei
Select the project, Test the mutation tool Milu, and Perform code mutation

Kang Yuwei
Match the test cases and mutants, Mark the location of the mutation, and Write the report

Reference

[1] https://github.com/yuejia/Milu

[2] https://github.com/nefuddos/Calculater

[3] Jia Y, Harman M. An Analysis and Survey of the Development of Mutation Testing[J]. IEEE Transactions on Software Engineering, 2011, 37(5):649-678.