



ISO2-2023-A04-Testing-P2

Implementation

Luis Eduardo Fernández-Medina Cimas

Anatoli Zournatz

Öykü Sedef Öztürk

Variable definition

Juan Alcázar Morales

Adrián Gómez del Moral Rodríguez Madridejos

Testing

Alejandro Del Hoyo Abad

Sergio Pozuelo Martín-Consuegra

1) Write, at least, the pseudocode of the identified method or methods. However, it will be desirable that you fully implement the code to have a better idea of the final code to be tested.

```
public class Complex {
    private double real;
    private double imaginary;

    public Complex(double real, double imaginary) {
        this.real = real;
        this.imaginary = imaginary;
    }

    public double getReal() {
        return real;
    }

    public double getImaginary() {
        return imaginary;
    }

    @Override
    public String toString() {
        return imaginary >= 0 ? real + " + " + imaginary + "i" : real + " - " + (-imaginary) + "i";
    }
}

public class QuadraticEquationSolver {
    public static Object[] calculateRoots(double a, double b, double c) {
        double discriminant = b * b - 4 * a * c;
        if (discriminant > 0) {
            double root1 = (-b + Math.sqrt(discriminant)) / (2 * a);
            double root2 = (-b - Math.sqrt(discriminant)) / (2 * a);
            return new Object[] { root1, root2 };
        } else if (discriminant == 0) {
            double root = -b / (2 * a);
            return new Object[] { root };
        } else {
            double realPart = -b / (2 * a);
            double imaginaryPart = Math.sqrt(-discriminant) / (2 * a);
            return new Object[] { new Complex(realPart, imaginaryPart), new Complex(realPart,
-imaginaryPart) };
        }
    }
}
```

}

2. Identify the variables that should be considered to test the method of interest.

In this case, we could test a, b and c, the coefficients of the quadratic equation that can be either real or complex.

3. Identify the test values for each previously identified variable mentioned above, specifying the technique used to obtain each of those values.

a: double

Equivalence classes:

Positive: $(0, \infty)$

Negative: $(-\infty, 0)$

Zero = 0

Testing values = {0, -1, -10, -0.3, -0.72, 1, 10, 0.3, 0.72}

b: double

Equivalence classes:

Positive: $(0, \infty)$

Negative: $(-\infty, 0)$

Zero = 0

Testing values = {0, -1, -10, -0.3, -0.72, 1, 10, 0.3, 0.72}

c: double

Equivalence classes:

Positive: $(0, \infty)$

Negative: $(-\infty, 0)$

Zero = 0

Testing values = {0, -1, -10, -0.3, -0.72, 1, 10, 0.3, 0.72}

Since equations can have values from negative infinity to positive infinity, it makes sense to separate them into either positive, zero or negative.

4. Calculate the maximum possible number of test cases that could be generated from the test values (combinatorics).

To calculate the maximum possible number of test cases that could be generated from the given test values of a, b and c, we can use the product of the number of values for each variable. Since each variable has 9 testing values, the total number of test cases can be calculated as follows:

Positive values:

Variations for $a*b*c = 4 * 4 * 4 = 64$

Zero:

Variations for $a*b*c = 1 * 1 * 1 = 1$

Negative values:

Variations for $a*b*c = 4 * 4 * 4 = 64$

Total:

The total number of test cases considering these assumptions for variations within each equivalence class would be the sum of the individual cases:

$64 \text{ (Positive values)} + 1 \text{ (Zero)} + 64 \text{ (Negative values)} = 129 \text{ test cases}$

5. Define a set of test cases to fulfill each use (each value once)

Considering the chosen testing values, we've chosen:

a: {0, -0.3, 0.72}

b: {1, 0.3, -0.72}

c: {-1, 10, -10}

Test suite = {(0, 1, -1), (-0.3, 0.3, 10), (0.72, -0.72, -10)}

6. Define test suites to achieve pairwise coverage using the proposed algorithm explained in the Lectures

Pairwise coverage is a test design technique that aims to test all possible pairs of input parameter values at least once. Because we can write infinity numbers, we had to work with a range instead.

Test Case	A	B	C
1	0	1	-1
2	0	0.3	10
3	0	-0.72	-10
4	-0.3	1	10
5	-0.3	0.3	-10
6	-0.3	-0.72	-1
7	0.72	1	-10
8	0.72	0.3	10
9	0.72	-0.72	1

7. For code segments that include decisions, propose a set of test cases to achieve coverage of decisions.

For the if discriminant, we could use this set of test cases:

- Positive discriminant = $\{(0, 1, -1), (0, 0.3, 10)\}$
- Zero discriminant = $\{(-0.3, 1, 10), (-0.3, 0.3, -10)\}$
- Negative discriminant = $\{(0.72, 1, -10), (0.72, 0.3, 10)\}$

8. For code segments that include decisions, propose a test case suite to achieve MC/DC coverage.

For the if discriminant, we could use this set of test cases:

- Positive discriminant = $\{(1, -1, 0), (-0.3, 0.3, 0.72)\}$
- Zero discriminant = $\{(0, 1, 0), (0, 0.72, -0.3)\}$
- Negative discriminant = $\{(1, 1, 1), (3, 0.72, 1)\}$
- False positive discriminant = $\{(0, 0, 0)\}$
- False zero discriminant = $\{(1, -0.3, 0.72)\}$
- False negative discriminant = $\{(-1, 0, 10)\}$
- Negate Condition for Positive Discriminant = $\{(-0.72, -0.3, 1)\}$
- Negate Condition for Zero Discriminant = $\{(-1, 1, 10)\}$
- Negate Condition for Negative Discriminant = $\{(0.72, 1, 0)\}$




9. Comment on the results of the number of test cases obtained in sections 4, 5, and 6, as well as the execution of the oracles: what can be said about the coverage achieved?

In the examination of testing strategies, three critical sections were explored: defined test cases (section 5), pairwise coverage (section 6), and decision coverage (section 7).

In section 4 we obtained 129 tests, a way too large amount to feasibly implement, so instead we designed 2 test methods. The first one we debated among sections 5 and 6, both consisting of 9 but ultimately we decided on implementing the test cases detailed in section 6 with pairwise coverage. For the second method, we debated among either section 7, made of 6 test cases, or section 8, made up of 14 test cases, but eventually we decided to use half of the test cases of section 8, the MC/DC coverage test cases.

Moreover, with respect to the coverage after being tested with Junit, the report with jacoco shows us that the coverage is of 62%. This is mainly because methods from the App and UserInterface class were not completely tested. However, the methods from the QuadraticEquationSolver are completely covered as expected.

Testing-P2-ISO2-2023-A04

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods	Missed Classes
 org.teamA04.iso		62 %		75 %	6 16	18 44	5 12	1 4
Total	98 of 263	62 %	2 of 8	75 %	6 16	18 44	5 12	1 4

org.teamA04.iso

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods	Missed Classes
Complex	<div><div></div><div></div></div>	29 %	<div><div></div><div></div></div>	0 %	2 5	1 7	1 4	0 1
App	<div><div></div><div></div></div>	0 %		n/a	2 2	8 8	2 2	1 1
UserInterface	<div><div></div><div></div></div>	64 %	<div><div></div><div></div></div>	100 %	1 5	8 17	1 4	0 1
QuadraticEquationSolver	<div><div></div><div></div></div>	97 %	<div><div></div><div></div></div>	100 %	1 4	1 12	1 2	0 1
Total	98 of 263	62 %	2 of 8	75 %	6 16	18 44	5 12	1 4