**SE Tools Exercise**

Team 8

**Contents**

**The program used 3**

**UML : Star UML 3**

**Results 3**

**Discussion 4**

**Unit Test : JUnit 5 5**

**Results 5**

**Discussion 7**

**Source Codes 8**

**The program used**

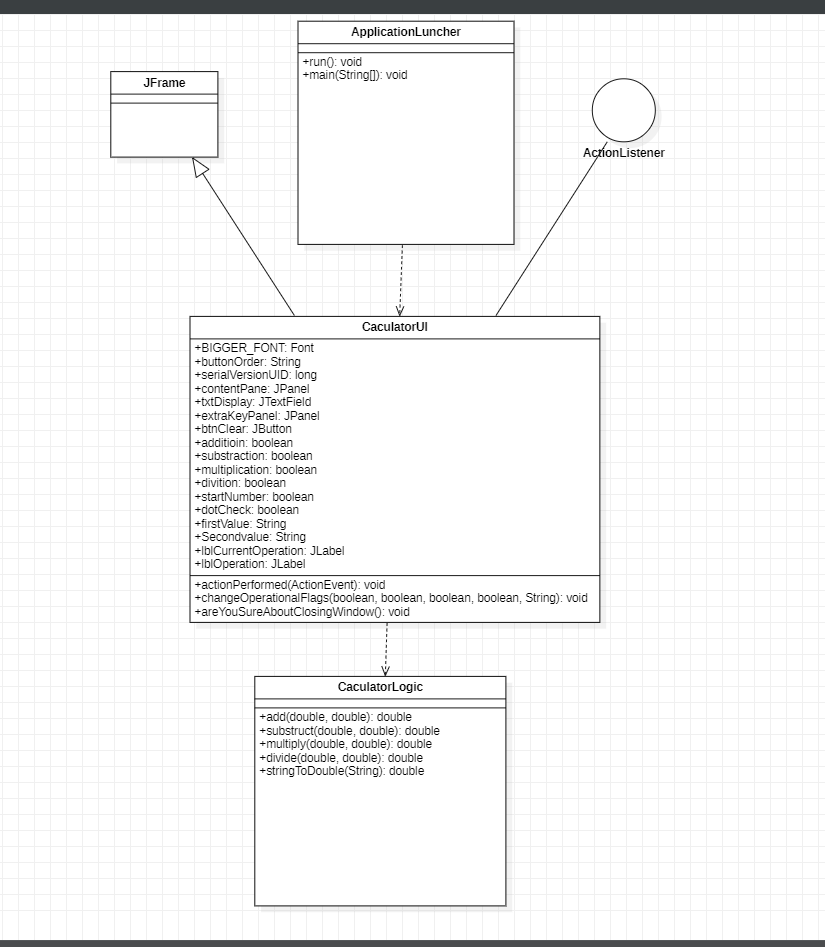
open-sourced codes : <https://github.com/rokon12/Simple-Calculator.git>

테이블이(가) 표시된 사진

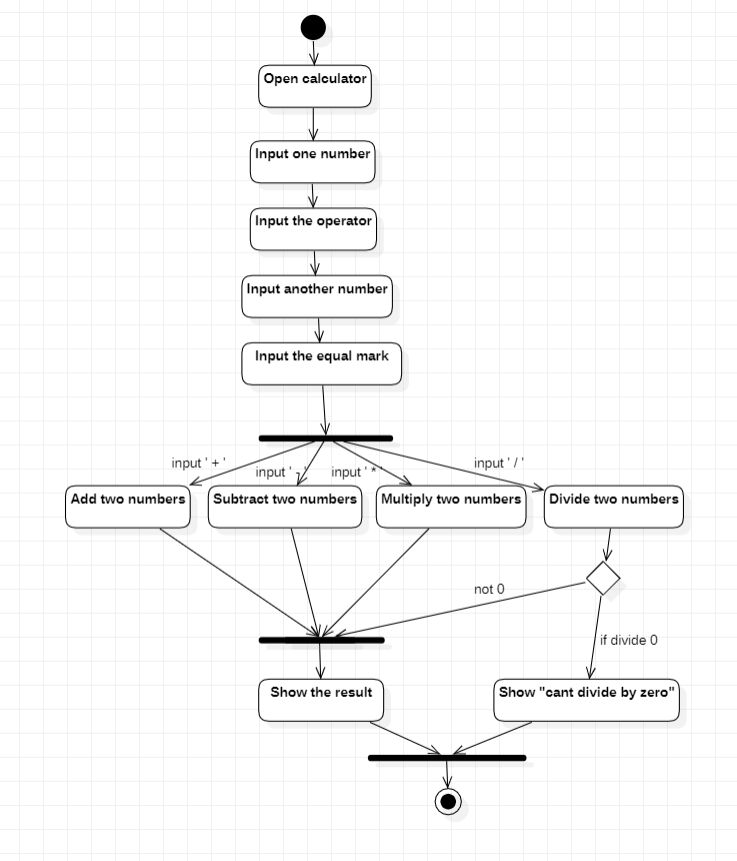
자동 생성된 설명

**UML: Star UML**

**Results**



**Class diagram**



**Activity diagram**

**Discussion**

**Pros**

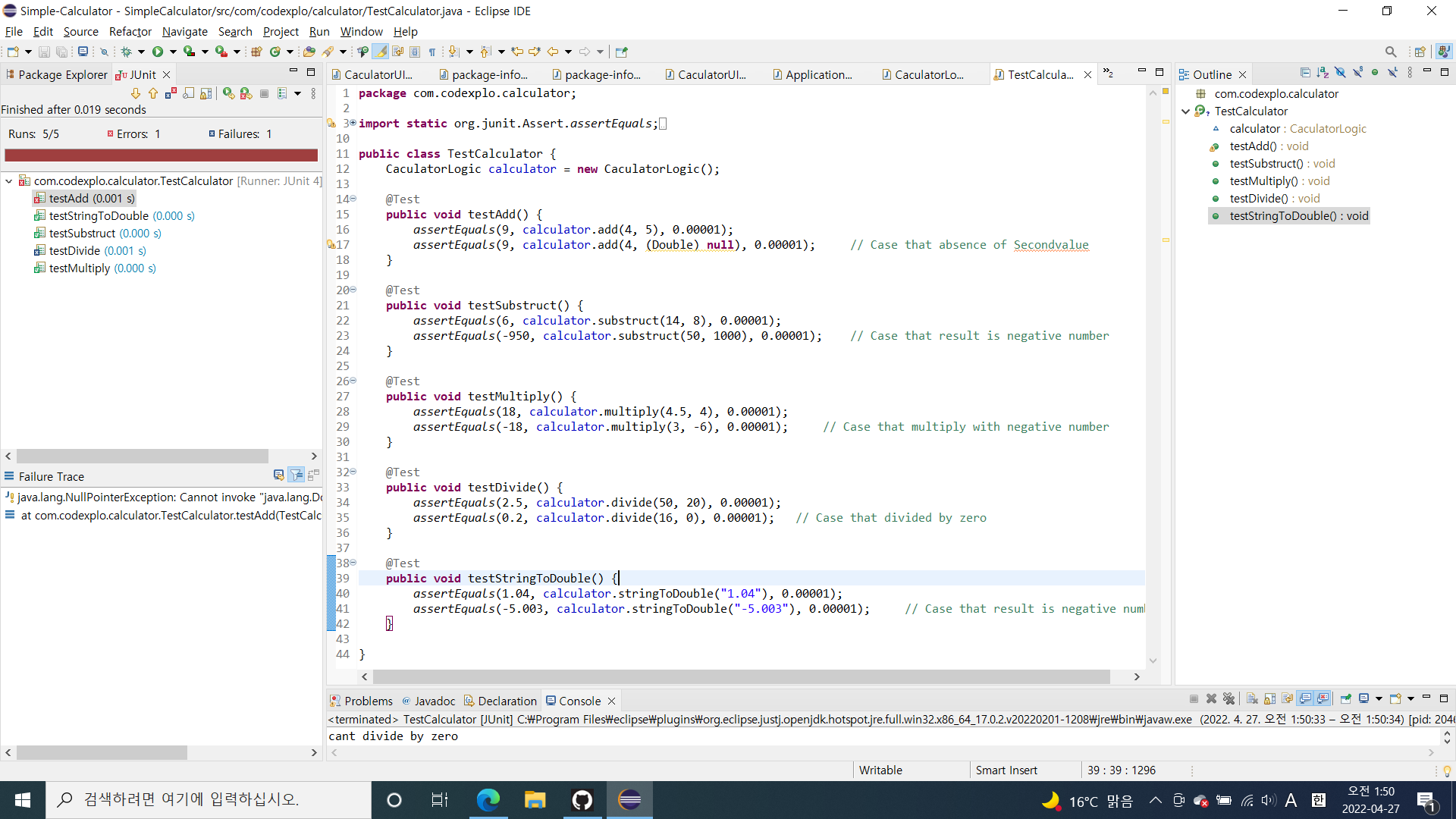
Codes that may seem difficult for non-majors to see are easily understood and expressed in drawings and texts familiar to everyone, so it is easier to understand and faster to deliver.

**Cons**

Since there are various ways of expression and the interface of various disciplines, it can be easy to draw but difficult to apply. Also, if it is difficult to implement as designed, the designed design can be ignored, and the use of uml increases the number of documents to be managed.

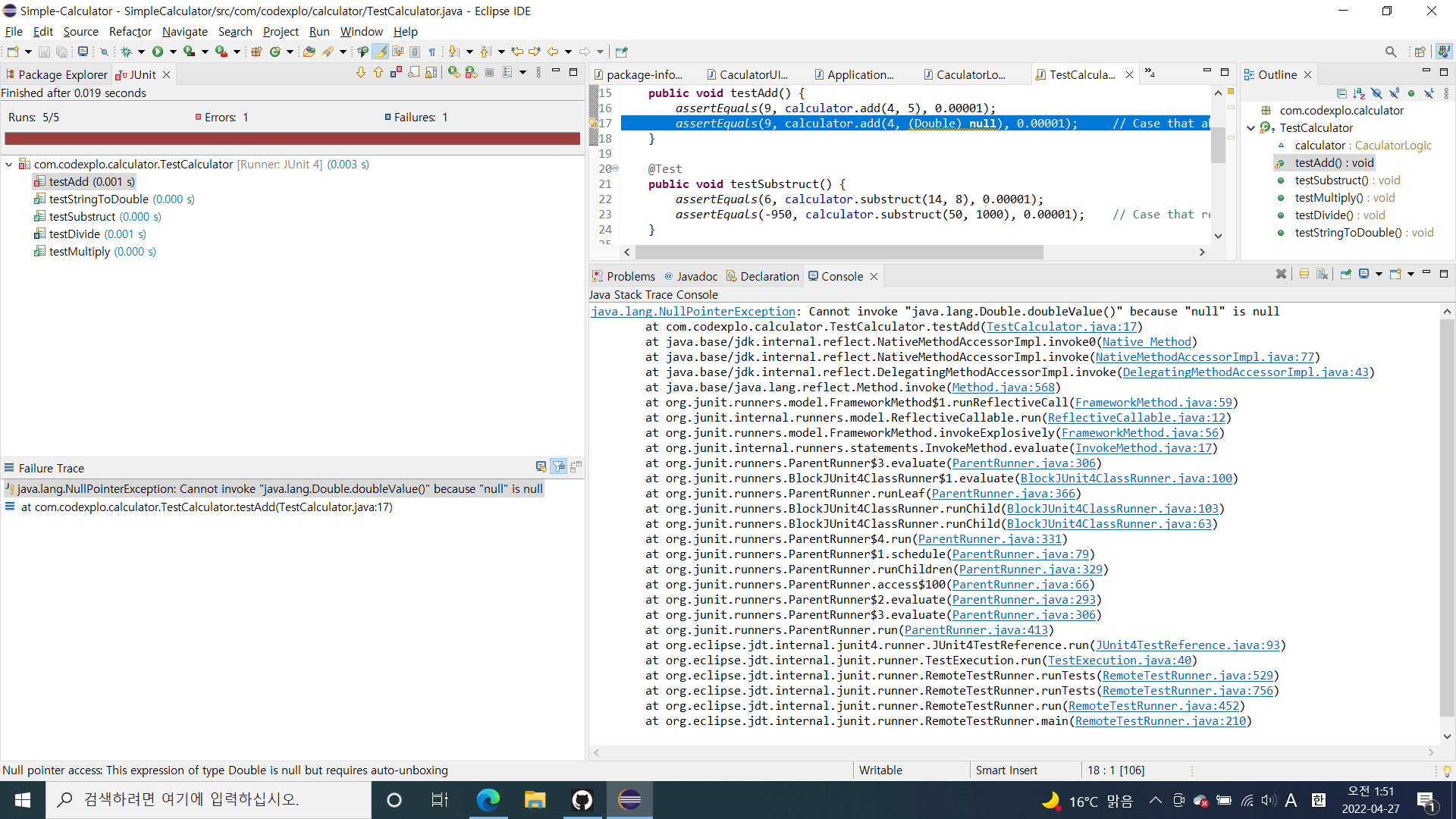
**Unit Test : JUnit 5**

**Results**



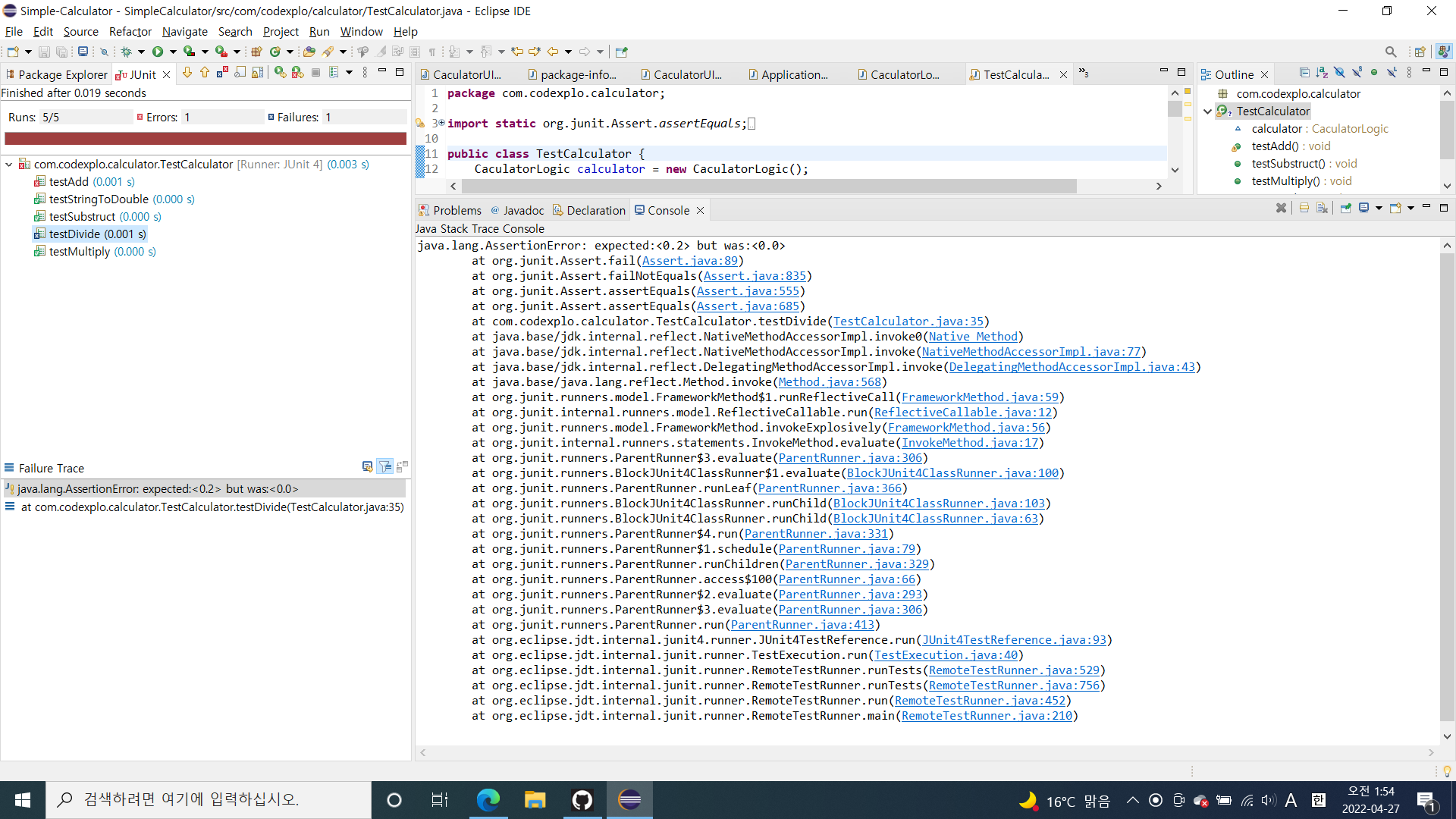
Tested 10 cases in 5 methods of CaculatorLogic.java.

1 Error and 1 Failure is detected.



Error : java.lang.NullPointerException: Cannot invoke "java.lang.Double.doubleValue()" because "null" is null

* Error occur when user click “=” after operator without second Value.
* So We should fix it.



Failure : java.lang.AssertionError: expected:<0.2> but was:<0.0>

* Divided by 0 Error handling with if… else…

**Solution for Error**

텍스트, 스크린샷, 노트북이(가) 표시된 사진

자동 생성된 설명

Add below codes (184-187) :

if (txtDisplay.getText().equals(""))

return;

And We opened pull request And It is Merged!

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트, 스크린샷, 실내, 모니터이(가) 표시된 사진

자동 생성된 설명

**Discussion**

**Pre**

**Reduced test time increases the likelihood of finding problems and improves reliability**

The purpose of unit tests is to isolate each part of the program and verify that each part works correctly. This allows you to quickly determine exactly which part went wrong in the event of a problem. Therefore, the stability of the program increases. Although unit tests seem to increase development time, they allow for leisurely programming by reducing the debugging time, which is the majority of the development period.

**Easier code changes because older features are also tested**

Programmers can trust unit tests and refactor them. After refactoring, you can be sure that the module is working as intended through unit tests. This is called regression testing. No matter how the code is fixed, the problem can be quickly identified and it is easy to know whether the modified code is working correctly, so that programmers can change the code more voluntarily.

**Simplified integration due to low coupling between tests**

Unit tests are useful in bottom-up testing because they remove the uncertainty of the unit itself. First, each part of the program is verified, and the parts are combined and verified again, which shines even more in the integration test.

**Cons**

Test execution speed is very slow. Manpower consumption is high, instability is high, and high costs are incurred.

**Source Codes**

**Full Codes** : <https://github.com/rokon12/Simple-Calculator.git>

**Testing Codes** :

package com.codexplo.calculator;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

import org.junit.jupiter.api.AfterAll;

import org.junit.jupiter.api.AfterEach;

import org.junit.jupiter.api.BeforeAll;

import org.junit.jupiter.api.BeforeEach;

public class TestCalculator {

CaculatorLogic calculator = new CaculatorLogic();

@Test

public void testAdd() {

assertEquals(9, calculator.add(4, 5), 0.00001);

assertEquals(9, calculator.add(4, (Double) null), 0.00001); // Case that absence of Secondvalue

}

@Test

public void testSubstruct() {

assertEquals(6, calculator.substruct(14, 8), 0.00001);

assertEquals(-950, calculator.substruct(50, 1000), 0.00001); // Case that result is negative number

}

@Test

public void testMultiply() {

assertEquals(18, calculator.multiply(4.5, 4), 0.00001);

assertEquals(-18, calculator.multiply(3, -6), 0.00001); // Case that multiply with negative number

}

@Test

public void testDivide() {

assertEquals(2.5, calculator.divide(50, 20), 0.00001);

assertEquals(0.2, calculator.divide(16, 0), 0.00001); // Case that divided by zero

}

@Test

public void testStringToDouble() {

assertEquals(1.04, calculator.stringToDouble("1.04"), 0.00001);

assertEquals(-5.003, calculator.stringToDouble("-5.003"), 0.00001); // Case that result is negative number

}

}