**Unit Test and Refactoring Questions**

1. Write a program using TDD to do the following (15 marks):

a. Return the gradient, distance, and equation of the line when the user enters two coordinates.

b. Refactor your code

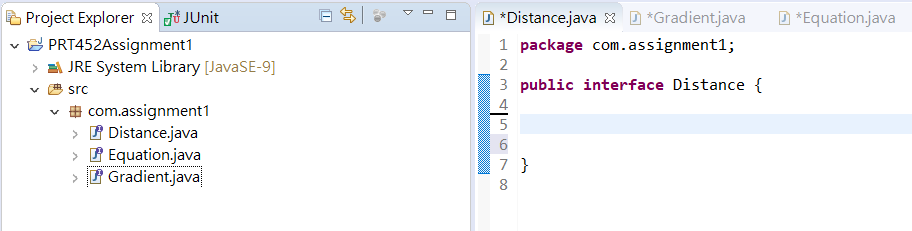
c. Create a Git directory for your assignment (including word or pdf documents and programming code)

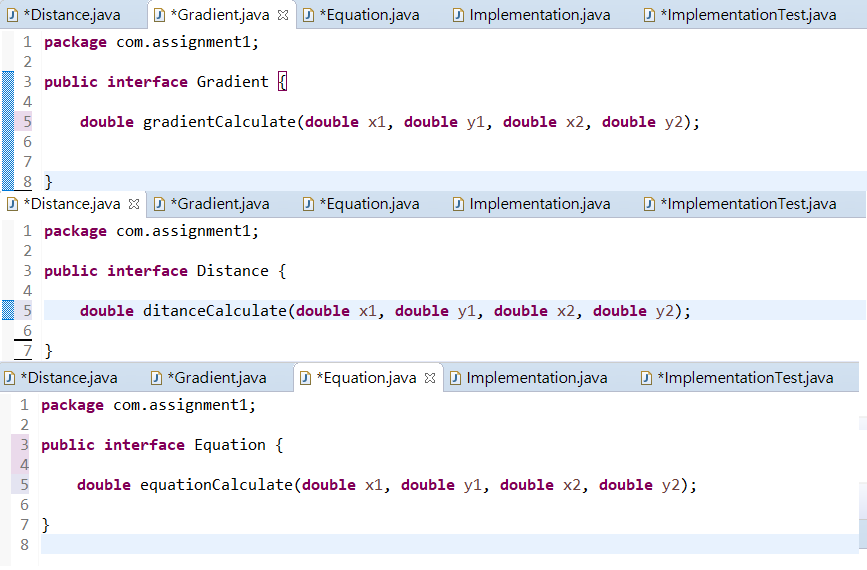
https://github.com/haihau/PRT452-Assignment-1

d. Capture screenshots and write briefly the steps we have taken to create this program

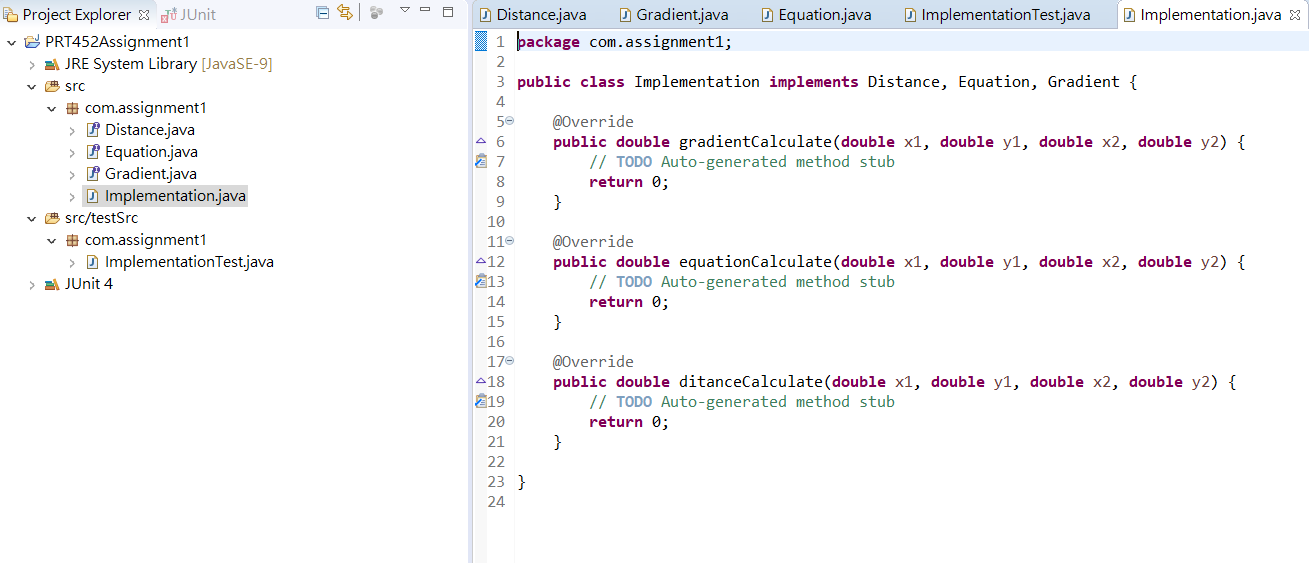
a.

**Step 1.** Create interfaces for gradient, distance, and equation.

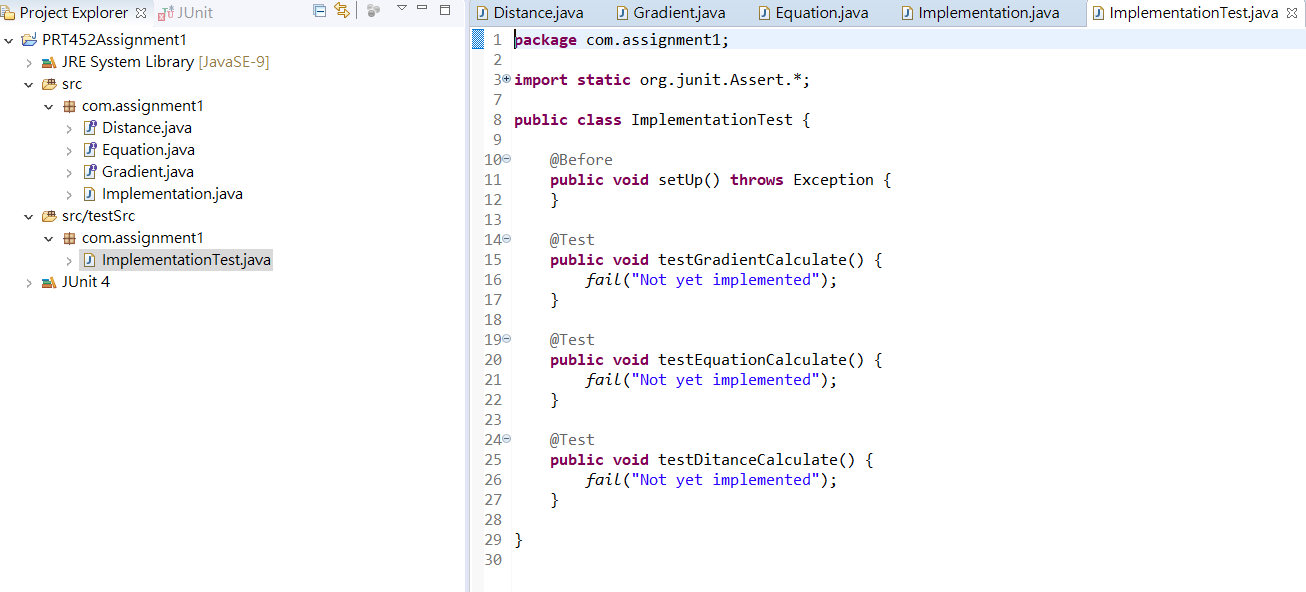


**Step 2.** Define each interface asgradientCalculate, distanceCalculate, and equationCalculate, for gradient, distance, and equation****

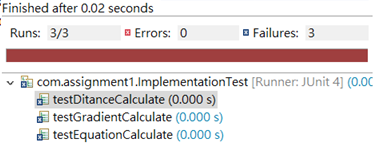
**Step 3.** Create a class name implementation which is do actual calculate for gradient, distance, and equation.

****

**Step 4.** Create a folder named testSrc for test case for the reason keep code clear, and then create a Junit test case named ImplementationTest undet the testSrc folder to write test cases for gradient, distance, and equation.

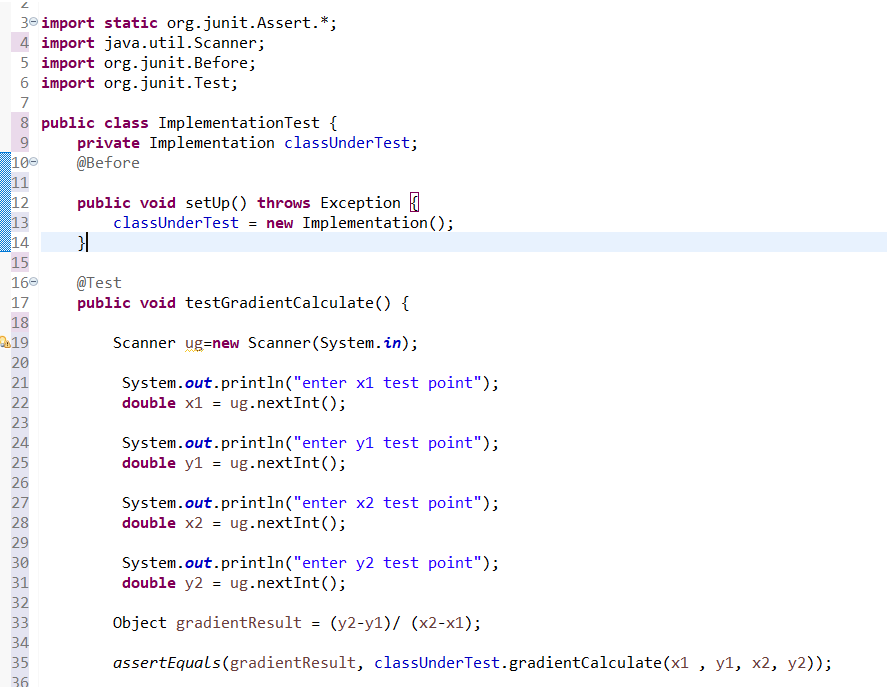
****

**Step 5.** Run ImplementationTest and expecting it will fail because we haven’t write any code yet.

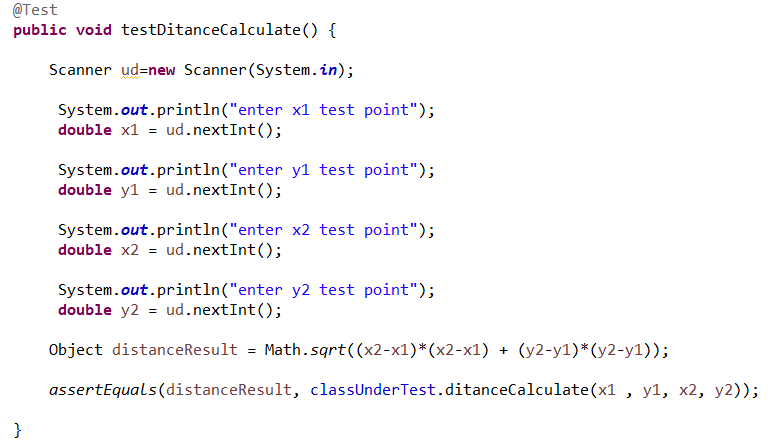
****

**Step 6.** Write test cases for testgradientCalculate, testdistanceCalculate, and testequationCalculate before we write main function.

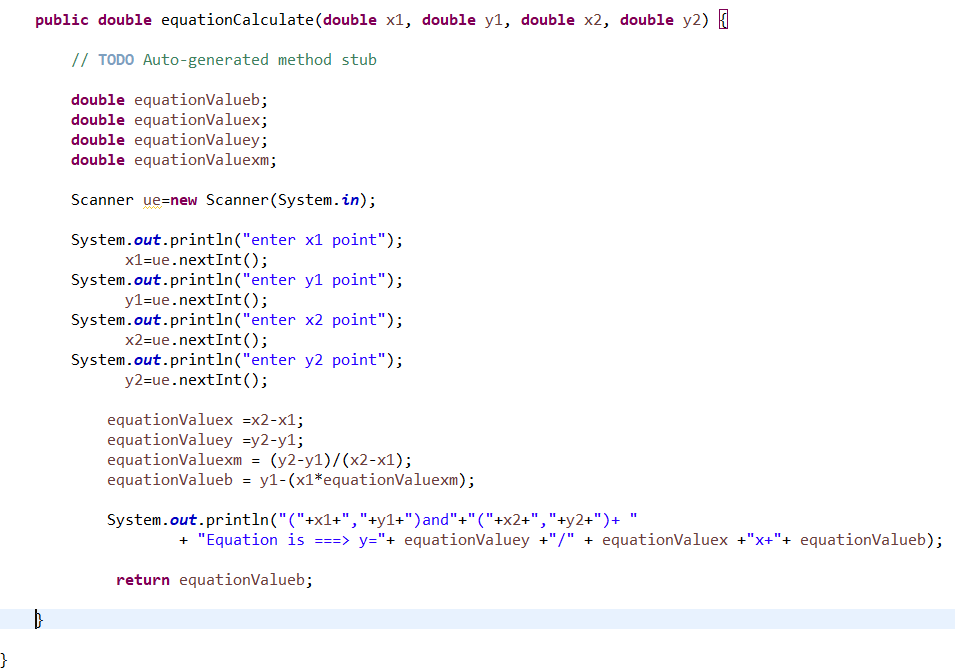
1. testgradientCalculate: first create a private variable named classUnderTest and start to write code for testgradientCalculate test case. What I did is catch the input x1, y1, x2, y2 from user and calculate the gradient. Also, use assertEquals function to compare the result from testgradientCalculate and from gradientCalculate.



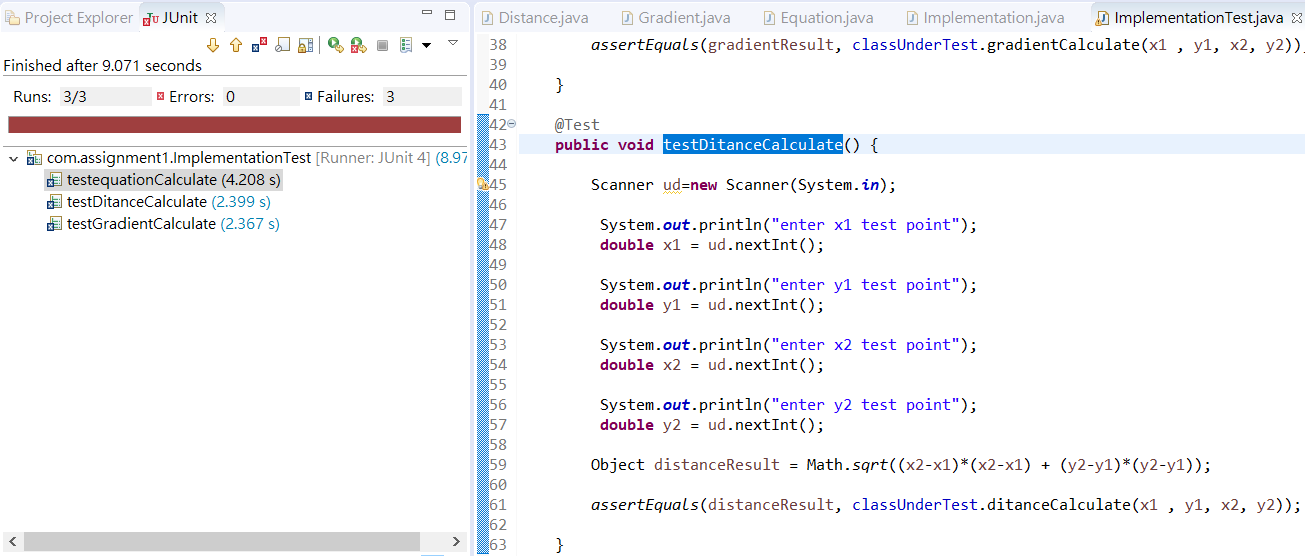
1. testdistanceCalculate: The same write code for testdistanceCalculate test case and use assertEquals function to compare the result from testdistanceCalculate and from distanceCalculate.



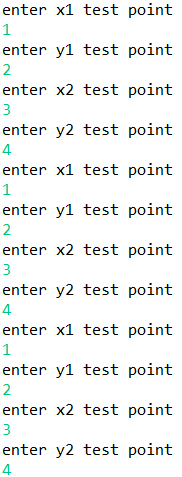
1. testequationCalculate: Write code for testequationCalculate test case and use assertEquals function to compare the value b from testequationCalculate and from quationCalculate because the formula is y=mx+b, and we have two coordinates, so we will know the value m. Instead of use the formula y=mx+b, I changed the formula to b=y-mx, so we just need to compare value b to test if the result is correct.



**Step 6.** Completed the test case and run the Junit test again and expect it will fail because we haven’t write the actual code for gradientCalculate, distanceCalculate, and equationCalculate.



I have to input x1, y1, x2, y2 for the testcase.

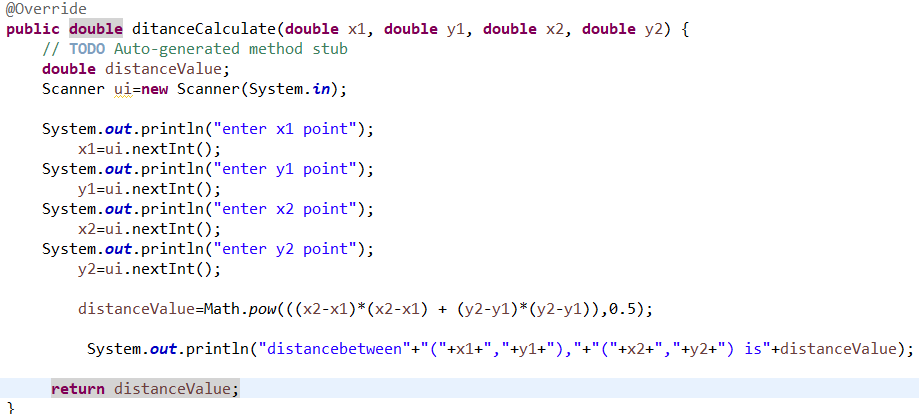


**Step 7.** Write the actual code for gradientCalculate, distanceCalculate, and equationCalculate.

1. gradientCalculate: I have changed the way of calculating, but the result should be the same.



1. distanceCalculate: I have changed the way of calculating, but the result should be the same.

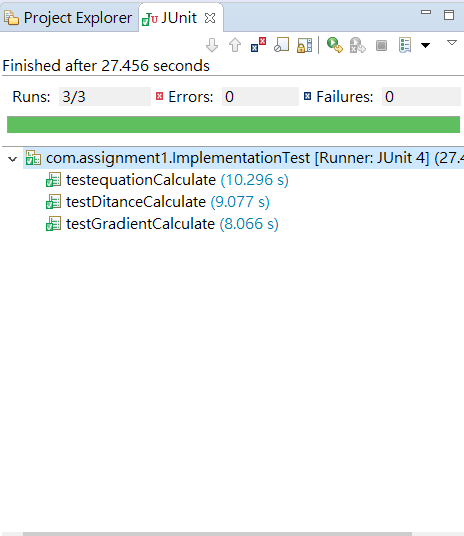


1. equationCalculate: I have changed the way of calculating, but the result should be the same.

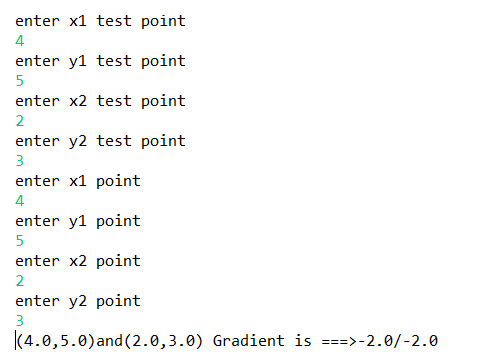


**Step 8.** Run the Junit test for gradientCalculate, distanceCalculate, and equationCalculate., and this time should get three passes because we complete the code.

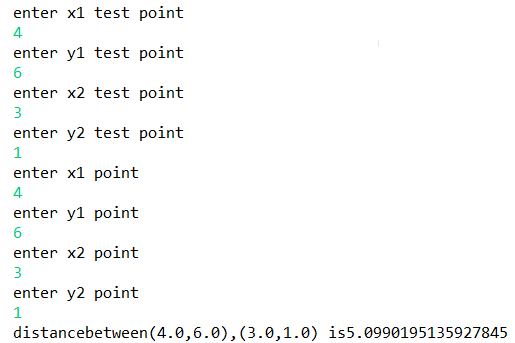
Result of test.



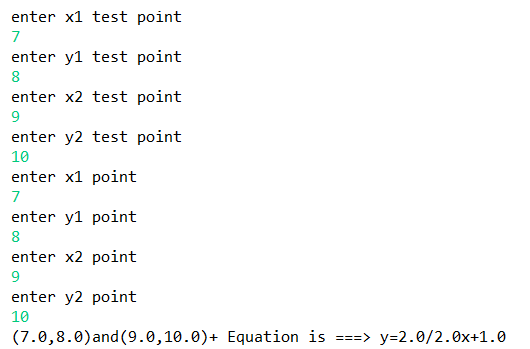
*Input and output for gradientCalculate.*



*Input and output for distanceCalculate.*



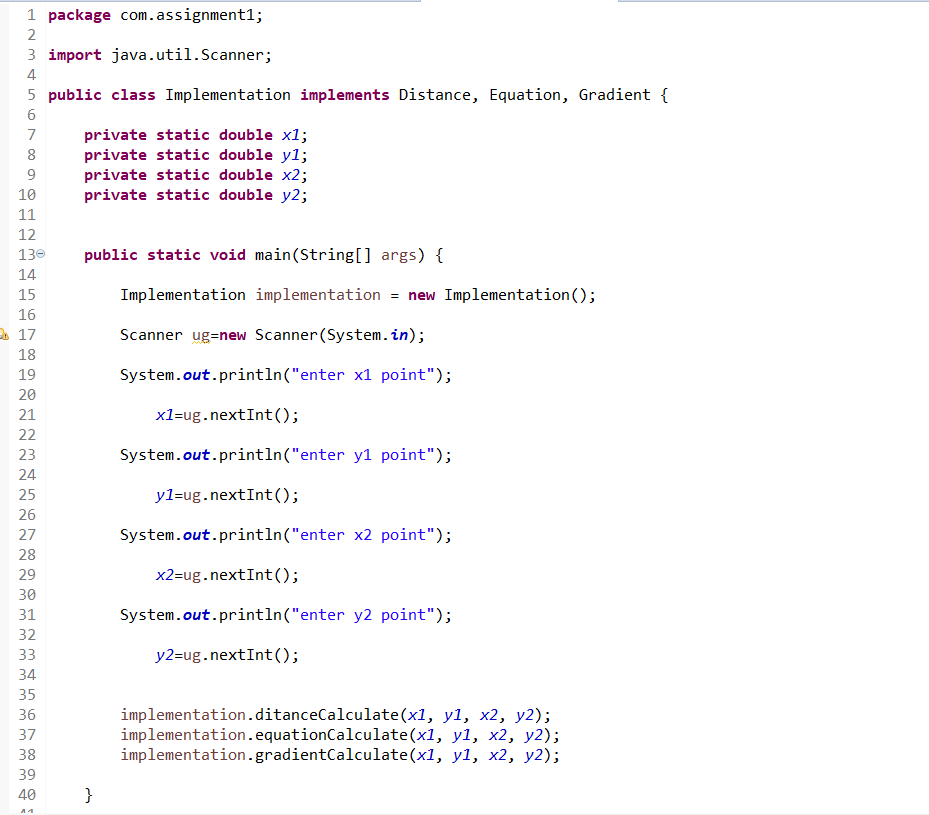
*Input and output for equationCalculate.*



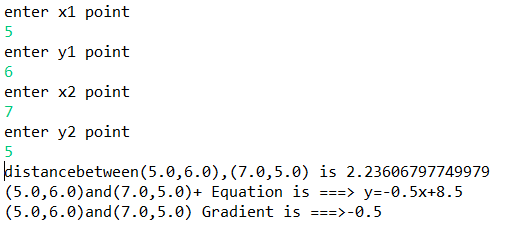
My program is done by TDD because before I write any code, I write test case first, so when I competed my code I do not need to check it again. Somehow, this method saved my time for finding bug and code mistakes.

b. Refactor code

1. Create a class for input x1, y1, x2, y2 for main function for the reason that reduce the number of user’s input. Meanwhile, test part still remain the same, so we can input diffident value in order to test those function separately.



*Result: user can input once and get three outcomes.*

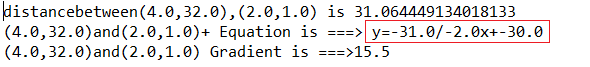


2. Create if else function in equationCalculate to display the value correctly.

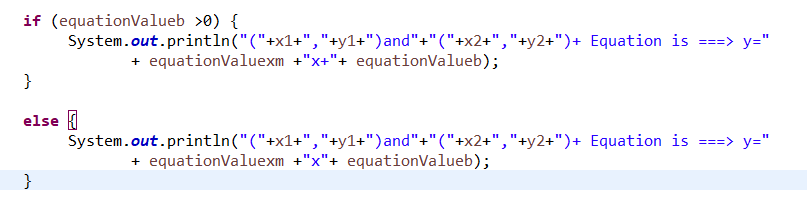
Before



Result



After



Result



**2. Code smell can give indications that there is some issue with the codes and can be solved by refactoring. Identify 5 issues and their related code smells. Include solutions to fix up the issues. (5 marks)**

**1) Duplicated Code**

The Duplicated Code is the most common code smell. If we see the same program structure in more than one location, we can try to combine them into one; the program will get better. Moreover, many other code smells are caused by Duplicated Code because repeated code often introduces bugs. When modifying or extending code, it must be consistently modified and missing one place will cause bugs.

(1) The first Duplicated Code is "the two functions of the same class contain the same function." In this case, all we need to do is extract the duplicate code by using the Extract Method and then use the code that we extract for both places.

(2) If two unrelated classes have Duplicated Code, we should consider using Extract Class on one of them, refining the duplicate code into a separate class, and then using the new class in another class. However, the duplicate code may indeed belong to only one class, another class can only call it, or the function may belong to the third class, and the other two classes should refer to the second class. We must decide this function which the best place to put it in, and make sure it doesn't appear anywhere else after it is placed.

**2) Large Class**

If we want to do a lot of things with a single class, there will often be too many instance variables in it. Once this happens, Duplicated Code will follow.

We can use the Extract Class to refine several variables together into a new class. When refining, we should select variables that are related to each other in the class and put them together. For example, depositAmount and depositCurrency should probably belong to the same class. Usually, if several variables in a class have the same prefix or suffix, this means that there is an opportunity to refine them into a component. If this component is suitable as a subclass, we will find that Extract Subclass is often simpler.

Sometimes classes don't use all variables. If so, we might be able to use Extract Class or Extract Subclass multiple times.

Like "too many variables," if there is too much code in the class, it is also the source of code duplication, confusion, and eventually hard to use. The simplest solution is to simple things are repeated inside the class. If there are five hundred-lines of functions, and many of them are the same function, then we can turn them into five ten-line functions and ten double-line functions.

As with "having too many variables," a class that has too much code is often suitable for using Extract Class and Extract Subclass.

**3) Switch Statements**

One of the most obvious features of object-oriented programs is the use of switch (or case) statements. The problem with switch statements is repetition. We will often find that the same switch statements scattered at different locations. If we want to add a new case clause to it, we must find all the switch statements and modify them. The polymorphic concept in object-oriented can be the solution to this.

Most of the time, as soon as we see the switch statement, we should consider replacing it with polymorphism. Switch statements are often chosen based on the type of code. We want "functions or classes related to the type of code," so we should use the Extract Method to refine the switch statement into a separate function, and then use the Move Method to move it into the class we needed. Sometimes, we must decide whether to use Replace Type Code with Subclasses or Replace Type Code with State/Strategy. Once we have completed the inheritance structure, we can use Replace Conditional with Polymorphism.

**4) Long Method**

If code too long is difficult to understand because it can cause inconsistencies, and it is not easy to use. When it required more data, we need to modify all the code. If we pass an object to a function, most of the modifications will be necessary, because we will most likely need to add one or two requests into the function to get more data.

If we can replace a parameter with a request to an existing object, then we should use the activate refactoring method "Replace the Parameter with a Method." Also, the "existing object" may be a field within the class to which the function belongs, or it may be referred to another parameter. We can also use "Preserve Whole Object" to collect a bunch of data from the same object and replace them. If some data lacks reasonable object attribution, we can use the "Introduce Parameter Object" to create a "parameter object" for them.

**5) Data Clumps**

Data items are like kids, and they like to stay together in groups. We can frequently see the same three or four peace of code in many places: the same fields in two classes, the same parameters in many feature signatures. Data that is constantly tied collectively need to have their own objects. In this case, we first locate out where the information appears as fields and use Extract Class to extract them into a separate object. Then turn our attention to the function signature and use the "Introduce Parameter Object" or "Preserve Whole Object" to reduce code for it. The immediate benefit of doing this is that you can shorten many lines of code and simplify functions. We don't need to worry about Data Clumps only use a part of the new object field because we can replace the two (or more) fields with new objects.

A good way to judge is to delete one of the many codes. Do this, do other codes lose meaning? If they don't make sense anymore, this is a clear signal: we should generate a new object for them.

Reducing the number of fields and codes can remove some code smell, but more importantly: Once we have a new object, we could let the program no longer smell. After we get the new object, we can start looking for "Feature Envy," and it can help us point out the procedural behaviours that can move to the new class. In the end, it doesn't take too long for all classes to be fully valued in the whole program.

# References

Van Emden, E., & Moonen, L. (2002). Java quality assurance by detecting code smells. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on* (pp. 97-106). IEEE.

Khomh, F., Di Penta, M., & Gueheneuc, Y. G. (2009, October). An exploratory study of the impact of code smells on software change-proneness. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on* (pp. 75-84). IEEE.

Sjoberg, D. I., Yamashita, A., Anda, B. C., Mockus, A., & Dyba, T. (2013). Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering*, (8), 1144-1156.

Olbrich, S., Cruzes, D. S., Basili, V., & Zazworka, N. (2009, October). The evolution and impact of code smells: A case study of two open source systems. In *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on* (pp. 390-400). IEEE.

Olbrich, S. M., Cruzes, D. S., & Sjøberg, D. I. (2010, September). Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In *Software Maintenance (ICSM), 2010 IEEE International Conference on* (pp. 1-10). IEEE.

Yamashita, A., & Moonen, L. (2013, October). Do developers care about code smells? an exploratory survey. In *Reverse Engineering (WCRE), 2013 20th Working Conference on* (pp. 242-251). IEEE.

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., & De Lucia, A. (2014, September). Do they really smell bad? a study on developers' perception of bad code smells. In *Software maintenance and evolution (ICSME), 2014 IEEE international conference on* (pp. 101-110). IEEE.

Carneiro, G. D. F., Silva, M., Mara, L., Figueiredo, E., Sant'Anna, C., Garcia, A., & Mendonca, M. (2010, September). Identifying code smells with multiple concern views. In *Software Engineering (SBES), 2010 Brazilian Symposium on* (pp. 128-137). IEEE.

Fontana, F. A., Mariani, E., Mornioli, A., Sormani, R., & Tonello, A. (2011, March). An experience report on using code smells detection tools. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on* (pp. 450-457). IEEE.