SOFTWARE ENGINEERING: PL2

Testing Code

Implementing and testing a mathematical function in Java

Lucas Álvarez Rodríguez-David Silveira Ángel
Group 1|2020-2021|GII UAH

INDEX

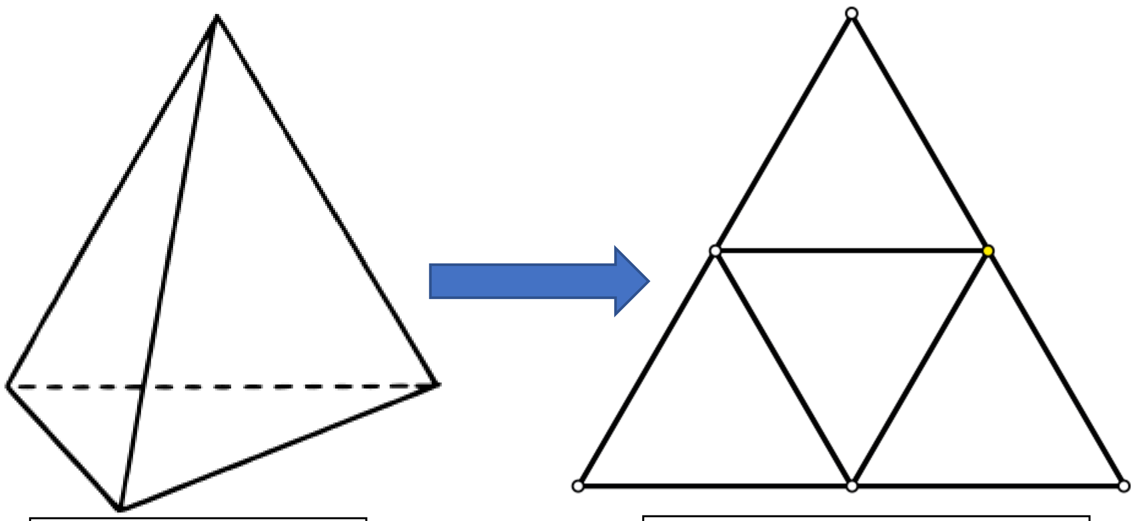
1. Implementing the function

For this practice, we have to implement a mathematical function in order to test it. We chose to implement the volume of an equilateral tetrahedron, as well as the combined surface area of all of its faces.

To do this, we divided the code into two parts: A 2D section and a 3D section.

We did this because it allows us to conceptualize the figure and therefore improve the comprehensibility of the program in case of any further changes along the way.

As we all know, the equilateral tetrahedron is composed of 4 equilateral triangles.



Equilateral tetrahedron

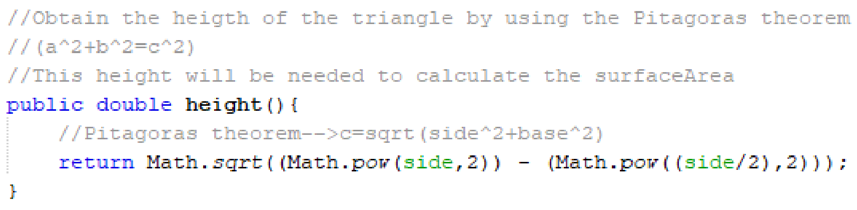
Four similar equilateral triangles

The only parameter needed for this function is the size of the side of the triangles, as all of them share said size.

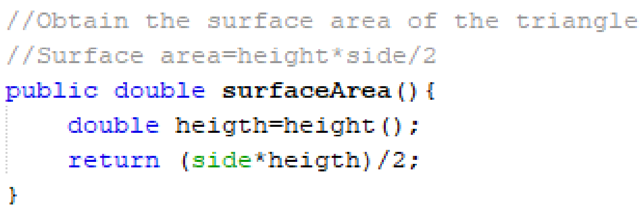
We implemented the function as it follows:

Triangle class:

We create a class called *Triangle* inside a *TwoDFigure* package. In it, we establish the methods needed to calculate the surface of each of the triangles, as well as the total surface area. In this class we have two methods: *height()* and *surfaceArea()*:



```
//Obtain the heigth of the triangle by using the Pitagoras theorem
//(a^2+b^2=c^2)
//This height will be needed to calculate the surfaceArea
public double height(){
    //Pitagoras theorem-->c=sqrt(side^2+base^2)
    return Math.sqrt((Math.pow(side,2)) - (Math.pow((side/2),2)));
}
```



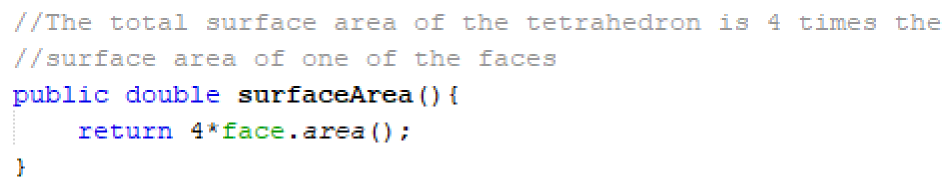
```
//Obtain the surface area of the triangle
//Surface area=height*side/2
public double surfaceArea(){
    double heigth=height();
    return (side*heigth)/2;
}
```

The 1st function is used as a simplification and modularization of the procedure of calculating the area of the triangles.

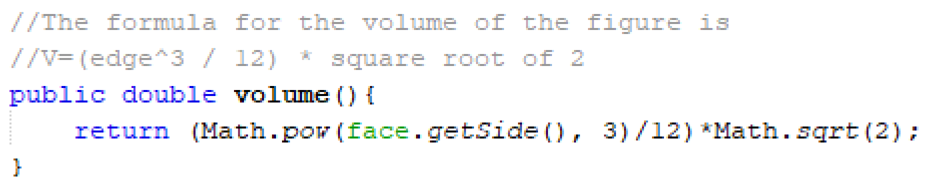
Tetrahedron class:

In the *Tetrahedron* class we make use of the previous *Triangle*'s methods to obtain the results we need for this class' own methods.

Here we can find *surfaceArea()* and *volume()*:



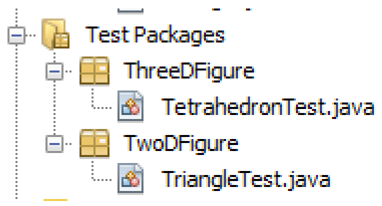
```
//The total surface area of the tetrahedron is 4 times the
//surface area of one of the faces
public double surfaceArea(){
    return 4*face.area();
}
```



```
//The formula for the volume of the figure is
//V=(edge^3 / 12) * square root of 2
public double volume(){
    return (Math.pow(face.getSide(), 3)/12)*Math.sqrt(2);
}
```

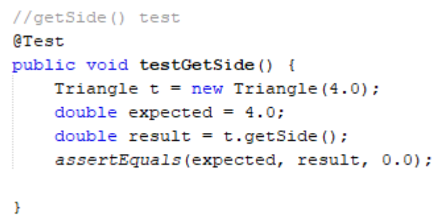
2. Junit testing

We now test the program by the use of Junit tests:

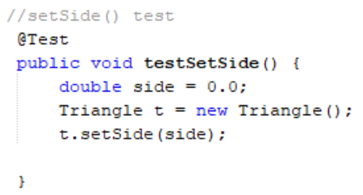


```
Test Packages
  ThreeDFigure
    TetrahedronTest.java
  TwoDFigure
    TriangleTest.java
```

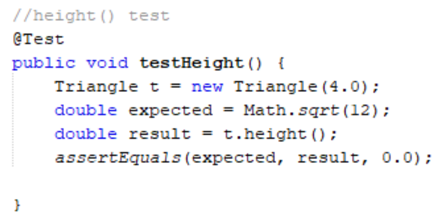
Triangle tests:



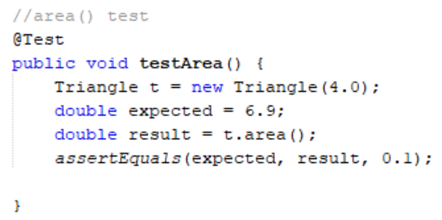
```
//getSide() test
@Test
public void testGetSide() {
    Triangle t = new Triangle(4.0);
    double expected = 4.0;
    double result = t.getSide();
    assertEquals(expected, result, 0.0);

}
```



```
//setSide() test
@Test
public void testSetSide() {
    double side = 0.0;
    Triangle t = new Triangle();
    t.setSide(side);

}
```

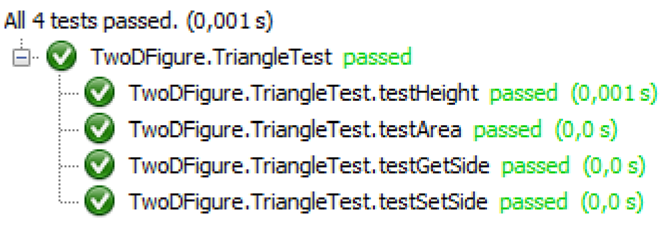


```
//height() test
@Test
public void testHeight() {
    Triangle t = new Triangle(4.0);
    double expected = Math.sqrt(12);
    double result = t.height();
    assertEquals(expected, result, 0.0);

}
```



```
//area() test
@Test
public void testArea() {
    Triangle t = new Triangle(4.0);
    double expected = 6.9;
    double result = t.area();
    assertEquals(expected, result, 0.1);

}
```

The results of these tests are checked by selecting 'Test file' and running them. The results are favorable:

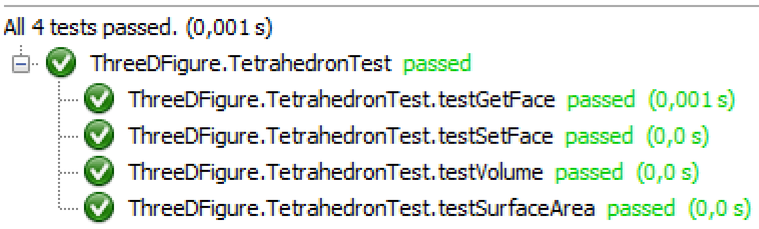


```
All 4 tests passed. (0,001 s)
TwoDFigure.TriangleTest passed
    TwoDFigure.TriangleTest.testHeight passed (0,001 s)
    TwoDFigure.TriangleTest.testArea passed (0,0 s)
    TwoDFigure.TriangleTest.testGetSide passed (0,0 s)
    TwoDFigure.TriangleTest.testSetSide passed (0,0 s)
```

Tetrahedron tests:

We do exactly the same for the *Tetrahedron* class:

```
Triangle triangle = new Triangle(4.0);
Tetrahedron tetrahedron = new Tetrahedron(triangle);

//getFace() test
@Test
public void testGetFace() {
    Triangle expected = triangle;
    Triangle result = tetrahedron.getFace();
    assertEquals(expected, result);

}

//setFace() test
@Test
public void testSetFace() {
    Triangle face = triangle;
    tetrahedron.setFace(face);

}

//surfaceArea() test
@Test
public void testSurfaceArea() {
    double expected = 27.6;
    double result = tetrahedron.surfaceArea();
    assertEquals(expected, result, 0.5);

}

//volume() test
@Test
public void testVolume() {
    double expected = 7.5;
    double result = tetrahedron.volume();
    assertEquals(expected, result, 0.5);

}
```



```
All 4 tests passed. (0,001 s)
ThreeDFigure.TetrahedronTest passed
    ThreeDFigure.TetrahedronTest.testGetFace passed (0,001 s)
    ThreeDFigure.TetrahedronTest.testSetFace passed (0,0 s)
    ThreeDFigure.TetrahedronTest.testVolume passed (0,0 s)
    ThreeDFigure.TetrahedronTest.testSurfaceArea passed (0,0 s)
```

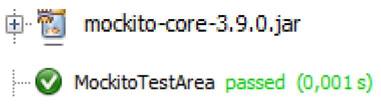
3. Exception

An obvious way to create an exception would be to consider the situation in which a negative number was inserted as the size of the side of the triangles. In this case, what we can do is set the value of the side parameter into an absolute number:

```
public Triangle (double side){
    this.side=Math.abs(side);
}
```

4. Mock test

For the mock test, we used *Mockito*. To demonstrate how to run a mock test with it, we employ the following code:

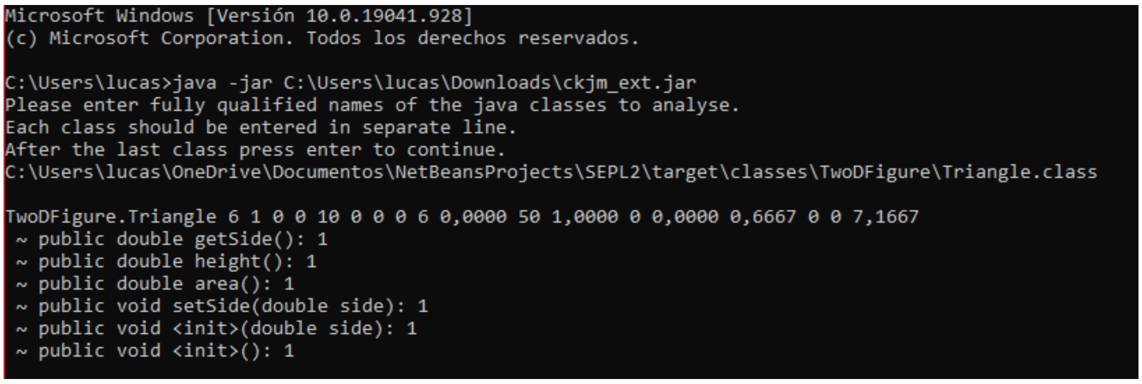


```
mockito-core-3.9.0.jar
MockitoTestArea passed (0,001 s)
```

```
//Mockito test for area()
@Test
public void MockitoTestArea(){
    Triangle tri = Mockito.mock(Triangle.class);
    tri.setSide(5);
    double result, expected=4.3301;

    Mockito.when(tri.area()).thenReturn(expected);

    result=tri.area();
    assertEquals(expected,result,0.0);
}
```

5. Code measure

To measure the code of the *Triangle* class we employ the CKJM tool. The result of using it is the following:



```
Microsoft Windows [Versión 10.0.19041.928]
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\lucas>java -jar C:\Users\lucas\Downloads\ckjm_ext.jar
Please enter fully qualified names of the java classes to analyse.
Each class should be entered in separate line.
After the last class press enter to continue.
C:\Users\lucas\OneDrive\Documentos\NetBeansProjects\SEPL2\target\classes\TwoDFigure\Triangle.class

TwoDFigure.Triangle 6 1 0 0 10 0 0 0 6 0,0000 50 1,0000 0 0,0000 0,6667 0 0 7,1667
 ~ public double getSide(): 1
 ~ public double height(): 1
 ~ public double area(): 1
 ~ public void setSide(double side): 1
 ~ public void <init>(double side): 1
 ~ public void <init>(): 1
```

The meaning of each value can be found in the following website:

http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/metric.html

The explanation of each number is, according to the page:

The metrics *ckjm* will calculate and display for each class are the following.

WMC - Weighted methods per class

A class's *weighted methods per class* WMC metric is simply the sum of the complexities of its methods. As a measure of complexity we can use the cyclomatic complexity, or we can abritrarily assign a complexity value of 1 to each method. The *ckjm* program assigns a complexity value of 1 to each method, and therefore the value of the WMC is equal to the number of methods in the class.

DIT - Depth of Inheritance Tree

The *depth of inheritance tree* (DIT) metric provides for each class a measure of the inheritance levels from the object hierarchy top. In Java where all classes inherit Object the minimum value of DIT is 1.

NOC - Number of Children

A class's *number of children* (NOC) metric simply measures the number of immediate descendants of the class.

CBO - Coupling between object classes

The *coupling between object classes* (CBO) metric represents the number of classes coupled to a given class (efferent couplings and afferent couplings). This coupling can occur through method calls, field accesses, inheritance, arguments, return types, and exceptions.

RFC - Response for a Class

The metric called the *response for a class* (RFC) measures the number of different methods that can be executed when an object of that class receives a message (when a method is invoked for that object). Ideally, we would want to find for each method of the class, the methods that class will call, and repeat this for each called method, calculating what is called the *transitive closure* of the method's call graph. This process can however be both expensive and quite inaccurate. In *ckjm*, we calculate a rough approximation to the response set by simply inspecting method calls within the class's method bodies. The value of RFC is the sum of number of methods called within the class's method bodies and the number of class's methods. This simplification was also used in the 1994 Chidamber and Kemerer description of the metrics.

LCOM - Lack of cohesion in methods

A class's *lack of cohesion in methods* (LCOM) metric counts the sets of methods in a class that are not related through the sharing of some of the class's fields. The original definition of this metric (which is the one used in *ckjm*) considers all pairs of a class's methods. In some of these pairs both methods access at least one common field of the class, while in other pairs the two methods to not share any common field accesses. The lack of cohesion in methods is then calculated by subtracting from the number of method pairs that don't share a field access the number of method pairs that do. Note that subsequent definitions of this metric used as a measurement basis the number of disjoint graph components of the class's methods. Others modified the definition of connectedness to include calls between the methods of the class. The program *ckjm* follows the original (1994) definition by Chidamber and Kemerer.

Ca - Afferent couplings

A class's afferent couplings is a measure of how many other classes use the specific class. Coupling has the same definition in context of Ca as that used for calculating CBO.

Ce - Efferent couplings

A class's efferent couplings is a measure of how many other classes is used by the specific class. Coupling has the same definition in context of Ce as that used for calculating CBO.

NPM - Number of Public Methods

The NPM metric simply counts all the methods in a class that are declared as public. It can be used to measure the size of an API provided by a package.

LCOM3 -Lack of cohesion in methods.

LCOM3 varies between 0 and 2.

m - number of procedures (methods) in class

a - number of variables (attributes in class

µ(A) - number of methods that access a variable (attribute)

$$\mathrm{LCOM}\mathbf{3} = \frac{\left(\frac{1}{a}\sum_{j=1}^{a}\mu(A_j)\right) - m}{1 - m}$$

The constructors and static initializations are taking into accounts as separately methods.

LOC - Lines of Code.

The lines are counted from java binary code and it is the sum of number of fields, number of methods and number of instructions in every method of given class.

DAM: Data Access Metric

This metric is the ratio of the number of private (protected) attributes to the total number of attributes declared in the class. A high value for DAM is desired. (Range 0 to 1)

MOA: Measure of Aggregation

This metric measures the extent of the part-whole relationship, realized by using attributes. The metric is a count of the number of data declarations (class fields) whose types are user defined classes.

MFA: Measure of Functional Abstraction

This metric is the ratio of the number of methods inherited by a class to the total number of methods accessible by member methods of the class. The constructors and the java.lang.Object (as parent) are ignored. (Range 0 to 1)

CAM: Cohesion Among Methods of Class

This metric computes the relatedness among methods of a class based upon the parameter list of the methods. The metric is computed using the summation of number of different types of method parameters in every method divided by a multiplication of number of different method parameter types in whole class and number of methods. A metric value close to 1.0 is preferred. (Range 0 to 1).

IC: Inheritance Coupling

This metric provides the number of parent classes to which a given class is coupled. A class is coupled to its parent class if one of its inherited methods functionally dependent on the new or redefined methods in the class. A class is coupled to its parent class if one of the following conditions is satisfied:

- One of its inherited methods uses a variable (or data member) that is defined in a new/redefined method.
- One of its inherited methods calls a redefined method.
- One of its inherited methods is called by a redefined method and uses a parameter that is defined in the redefined method.

CBM: Coupling Between Methods

The metric measure the total number of new/redefined methods to which all the inherited methods are coupled. There is a coupling when one of the given in the IC metric definition conditions holds.

AMC: Average Method Complexity

This metric measures the average method size for each class. Size of a method is equal to the number of java binary codes in the method.

CC - The McCabe's cyclomatic complexity

It is equal to number of different paths in a method (function) plus one.
The cyclomatic complexity is defined as:
$CC = E - N + P$
where
E - the number of edges of the graph
N - the number of nodes of the graph
P - the number of connected components