

DOCUMENTAZIONE DI TEST



NOME GRUPPO: The golden unicorns

Componenti:

Benedetta Totaro **matricola:** 735299 **e-mail:** b.totaro2@studenti.uniba.it

Giovanni Tedesco **matricola:** 745825 **e-mail:** g.tedesco16@ studenti.uniba.it

Piercamillo Turturro **matricola:** 736753 **e-mail:** p.turturro2@ studenti.uniba.it

Sommario

1. HOMEWORK 1	2
1.1. CODICE.....	2
1.2. UNDERSTANDING THE REQUIREMENTS	5
1.3. EXPLORE WHAT PROGRAM DOES FOR VARIOUS INPUT	5
1.4. EXPLORE INPUTS, OUTPUTS AND IDENTIFY PARTITIONS.....	6
1.5. IDENTIFY BOUNDARY CASES	8
1.6. DEVISE TEST CASE	9
1.7. AUTOMATE TEST CASES	10
1.8. MISSING SOMETHING?	15
1.9. FAILED TESTS	15
1.10. CHANGES AFTER FAILED TESTS	15
2. HOMEWORK 2	16
2.1. PRIMA PARTE	16
2.2. SECONDA PARTE	16
2.2.1. CODICE DA TESTARE.....	16
2.2.2. TEST PER RAGGIUNGERE IL 100% DI COVERAGE	17
2.2.3. BLACK BOX	18
3. HOMEWORK 3	23
3.1. CODICE.....	23
3.2. DEVISE TEST CASE	23
3.3. AUTOMATE TEST CASE	24
3.4. STATISTICHE	25

1. HOMEWORK 1

1.1. CODICE

CLASSE LISTMANAGER:

```
private List<Element> elements;
private float totElement=0;

public ListManager(List<Element> elements) throws Exception {
    if(elements == null){
        throw new NullPointerException("The list can't be null");
    }
}
```

```

        this.elements=elements;
    }

    public void addElement(double cost, int quantity, String productName) throws
    Exception{

        Element element=new Element(productName, cost, quantity);
        elements.add(element);

        System.out.println("The amount of " + element.returnProductName() + " is: "
+ element.returnTotElement()+"\n");
    }

    public double calculateTotalAmount() throws Exception {

        double tot=0;

        for(Element element:this.elements) {

            tot = tot + element.returnTotElement();

        }

        addTotalAmountToFile(tot);
        return tot;
    }

    public int removeElement(String productName){

        int removedElements=0;

        for(Element element:elements){

            if(element.returnProductName().equals(productName)){

                elements.remove(element);
                removedElements++;

            }

        }

        return removedElements;
    }

    private void addTotalAmountToFile(double tot) throws Exception{

        FileWriter writer = new
FileWriter("C:\\Users\\Bened\\OneDrive\\Desktop\\integrazione e
test\\homework\\homework1.csv");
        writer.append("Product Name,Cost,Quantity,Total amount=" +
String.valueOf(tot) + "\n");
        for(Element element:this.elements) {
            writer.append(element.returnProductName());
            writer.append(",");
            writer.append(String.valueOf(element.returnCost()));
            writer.append(",");
            writer.append(String.valueOf(element.returnQuantity()));

```

```

        writer.append(",");
        writer.append(String.valueOf(element.returnTotElement()));
        writer.append("\n");
    }

    writer.flush();
    writer.close();
}

public int returnListSize(){
    return elements.size();
}

```

CLASSE ELEMENT:

```

private String productName;
private double cost;
private int quantity;
private double totElement;

public Element(String productName, double cost, int quantity) {
    if (productName == null || productName.equals("") ||
    productName.equals("\n") || productName.equals("\t")) {
        throw new IllegalArgumentException("The product name can't be empty.");
    }

    if(cost < 0.01 || cost > 1000) {
        throw new IllegalArgumentException("The cost can't be negative or equal
to zero.");
    }

    if(quantity <= 0 || quantity > 1000) {
        throw new IllegalArgumentException("The amount of element can't be
negative or equal to zero.");
    }

    this.totElement = cost*quantity;
    this.productName = productName;
    this.cost = cost;
    this.quantity = quantity;
}

public double returnCost() {
    return this.cost;
}

```

```

public double returnTotElement() {
    return this.totElement;
}

public int returnQuantity() {
    return this.quantity;
}

public String returnProductName() {
    return this.productName;
}

```

1.2. UNDERSTANDING THE REQUIREMENTS

L'obiettivo del programma è salvare la lista dei prodotti, mostrare il costo totale di ogni singolo prodotto e, se richiesto, mostrare il costo totale di tutti gli elementi presenti nella lista.

Per ogni lista è possibile anche calcolare il numero totale degli elementi presenti nella lista.

Per ogni acquisto il programma il programma prende in input nome, prezzo e quantità del prodotto e li salva su un file Excel mostrando successivamente, il costo totale del prodotto appena registrato nel catalogo.

Nel caso in cui si decida di farlo, verrà calcolato il costo totale di tutti i prodotti acquistati e verrà salvato anch'esso sul file. Il costo non può essere più piccolo di 0,01 e maggiore di 1000, la quantità indicata non può essere minore o uguale di 0 né maggiore di 1000, il nome del prodotto non può essere nullo, vuoto un carattere di new line né di tabulazione.

1.3. EXPLORE WHAT PROGRAM DOES FOR VARIOUS INPUT

```

@Test
@DisplayName("Understand inputs of element")
void understandInputsOfElement() {
    Element element = new Element("pen", 2.50, 4);
    assertNotNull(element);
    assertEquals("pen", element.returnProductName());
    assertEquals(2.50, element.returnCost());
    assertEquals(4, element.returnQuantity());
}

@Test
@DisplayName("Understand inputs of list manager")
void understandInputsOfListManager() throws Exception{
    List<Element> elements = new ArrayList<>();
}

```

```

Element element=new Element("bananas",12.0,5);

elements.add(element);

element=new Element("strawberries",9.20,4);

elements.add(element);

element=new Element("apples",10.50,10);

elements.add(element);

ListManager listManager = new ListManager(elements);

assertNotNull(listManager);

listManager.addElement( 20,20,"pineapples");

listManager.addElement(10.02, 30,"pears");

Assertions.assertEquals(5,listManager.returnListSize());

}

```

1.4. EXPLORE INPUTS, OUTPUTS AND IDENTIFY PARTITIONS

Per ogni classe del programma sono stati analizzati inputs e outputs di ogni metodo.

CLASSE LISTMANAGER:

1. Metodo ListManager:

- Inputs:

List<Element> elements:

1. Null
2. Lunghezza>=0

- Outputs:

il metodo non fornisce elementi in output

2. Metodo calculateTotalAmount:

- Inputs:

il metodo non richiede elementi in input

- Outputs:

Float tot:

1. Totale>0
2. Totale<=1000000*numero di elementi presenti nella lista

3. Metodo addElement:

- Inputs:

Float cost:

1. Cost<=0
2. Cost>0
3. Cost<1000
4. Cost>=1000

Int quantity:

1. Quantity<=0
2. Quantity>0
3. Cost<1000
4. Cost>=1000

String productName:

1. Null
2. Empty
3. Lunghezza della stringa >= 1

- **Outputs:**

il metodo non restituisce elementi in output

4. Metodo removeElement:

- **Inputs:**

String productName:

1. Null
2. Empty
3. Lunghezza della stringa >= 1

- **Outputs:**

Int removedElements:

1. removedElement >= 0

Gli altri metodi della classe ListManager non sono stati presi in considerazione perché comunicano con un file, perciò, non fanno parte della tipologia di test che realizzeremo per l'homework 1.

CLASSE ELEMENT:

1. Metodo Element:

- **Inputs:**

Float cost:

1. Cost <= 0
2. Cost > 0
3. Cost < 1000
4. Cost >= 1000

Int quantity:

1. Quantity <= 0
2. Quantity > 0
3. Cost < 1000
4. Cost >= 1000

String productName:

1. Null
2. Empty
3. Lunghezza della stringa >= 1

- **Outputs:**

il metodo non fornisce elementi in output

2. Metodo returnCost:

- **Inputs:**
il metodo non richiede elementi in input
- **Outputs:**

Float cost:

1. Cost<=0
2. Cost>0
3. Cost<1000
4. Cost>=1000

3. Metodo returnTotElement:

- **Inputs:**
il metodo non richiede elementi in input
- **Outputs:**

Float totElement:

1. Totale>=0
2. Totale<1000000

4. Metodo returnQuantity:

- **Inputs:**
il metodo non richiede elementi in input
- **Outputs:**

Int quantity:

1. Quantity<=0
2. Quantity>0
3. Cost<1000
4. Cost>=1000

5. Metodo returnProductName:

- **Inputs:**
il metodo non richiede elementi in input
- **Outputs:**

String productName:

1. Null
2. Empty
3. Lunghezza della stringa>=1

1.5. IDENTIFY BOUNDARY CASES

Il team si è occupato di analizzare le variabili coinvolte in if-statement per identificare i boundary cases. Le variabili analizzate sono le seguenti:

1. quantity<=0 || quantity>1000

Left:

on point: 0

off point: 1

Right:

on point: 1000

off point: 1001

2. `cost < 0.01 || cost > 1000`

Left:

on point: 0.01

off point: 0.01 – Double.MIN_VALUE

Right:

on point: 1000

off point: 1000 + Double.MIN_VALUE

3. `productName == null || productName == ""`

on point: 0

off point: 1

1.6. DEVISE TEST CASE

Classe Element:

T1: productName null

T2: productName empty

T3: productName \n

T4: productName \t

T5: productName correct

T6: cost left on point

T7: cost left off point

T8: cost right on point

T9: cost right off point

T10: quantity left on point

T11: quantity left off point

T12: quantity right on point

T13: quantity right off point

T14: productName correct

T15: cost correct

T16: quantity correct

T17: list not null neither empty

Classe ListManager:

T1: list null

T2: list empty and with one element

T3: element correct and empty list

T4: element correct and full list (add)

T5: element correct and empty list and list without the element

T6: element correct and full list (remove)

T7: empty list (calculate total amount)

T8: full list (calculate total amount)

1.7. AUTOMATE TEST CASES

Classe ElementTest:

```
@Nested
@DisplayName("writing tests for the constructor")
class ConstructorTests {

    @Test
    @DisplayName("Product name null should return Exception")
    void productNameNullShouldReturnException() throws Exception {

        assertThrows(IllegalArgumentException.class, () -> new Element(null,2,3)
    );

    }

    @Test
    @DisplayName("Product name empty should return Exception")
    void productNameEmptyShouldReturnException() throws Exception {

        assertThrows(IllegalArgumentException.class, () -> new Element("",2,3) );

    }

    @Test
    @DisplayName("Product name new line should return Exception")
    void productNameNewLineShouldReturnException() throws Exception {

        assertThrows(IllegalArgumentException.class, () -> new Element("\n",2,3)
    );

    }

    @Test
    @DisplayName("Product name tab should return Exception")
    void productNameTabShouldReturnException() throws Exception {

        assertThrows(IllegalArgumentException.class, () -> new Element("\t",2,3)
    );

    }

    @Test
    @DisplayName("Product name correct should return a non null object")
```

```

void productNameCorrectShouldReturnAnNonEmptyObject() {

    Assertions.assertNotNull(new Element("ciao", 2.3, 3));

}

@Test
@DisplayName("Cost on point value (0.01) should return not null object")
void costLeftOnPointValueShouldReturnAnNonEmptyObject() throws Exception {

    assertNotNull(new Element("prova", 0.01, 3) );

}

@Test
@DisplayName("Cost off point value (0.01 - Float.MIN_VALUE) should return
exception")
void costLeftOffPointValueShouldReturnException() {

    assertThrows(IllegalArgumentException.class, ()-> new
Element("prova", Math.nextAfter(0.01, 0), 3));

}

@Test
@DisplayName("Cost right on point value (1000) should return a not null
object")
void costRightOnPointValueShouldReturnANotNullObject() throws Exception {

    assertNotNull(new Element("prova", 1000 , 3) );

}

@Test
@DisplayName("Cost right off point value (1000 + Double.MIN_VALUE) should
return exception")
void costRightOffPointValueShouldReturnException() {

    assertThrows(IllegalArgumentException.class, ()-> new
Element("prova", Math.nextAfter(1000, 1001), 3));

}

@Test
@DisplayName("Quantity left on point value(0) should return Exception")
void quantityLeftOnPointValueShouldReturnException() throws Exception {

    assertThrows(IllegalArgumentException.class, () -> new Element("prova",
20, 0));

}

@Test
@DisplayName("Quantity left off point value(1) should return a non null
object")
void quantityLeftOffPointValueShouldReturnANotNullObject() {

    Assertions.assertNotNull(new Element("prova", 20, 1));

}

@Test
@DisplayName("Quantity right on point value(1000) should return a not null

```

```

object")
    void quantityRightOnPointValueShouldReturnANotNullObject() throws Exception
    {

        assertNotNull(new Element("prova", 20, 1000));

    }

    @Test
    @DisplayName("Quantity right off point value(1001) should return exception")
    void quantityRightOffPointValueShouldReturnException() {

        assertThrows(IllegalArgumentException.class, ()->new
Element("prova",20,1001));

    }

}

@Test
@DisplayName("the method returnProductName should return the correct product
name")
void returnProductNameShouldReturnTheNameOfTheProduct () {

    Element element = new Element("ciao", 2.5F,3);

    Assertions.assertEquals("ciao", element.returnProductName());

}

@Test
@DisplayName("the method returnCost should return the correct cost of the
product")
void returnCostShouldReturnTheCorrectCostOfTheProduct () {

    Element element = new Element("ciao", 2.5F,3);

    Assertions.assertEquals(2.5F, element.returnCost());

}

@Test
@DisplayName("the method returnQuantity should return the correct quantity of
the product")
void returnQuantityShouldReturnTheCorrectQuantityOfTheProduct () {

    Element element = new Element("ciao", 2.5F,3);
    Assertions.assertEquals(3, element.returnQuantity());

}

@Test
@DisplayName("the method returnTotElement should return the correct quantity of
the product")
void returnTotElementShouldReturnTheCorrectTotalOfTheProduct () {

    Element element = new Element("ciao", 2.5F,3);
    Assertions.assertEquals((2.5F * 3), element.returnTotElement());

}

```

Classe ListManagerTest:

```
@Nested
@DisplayName("writing tests for the constructor")
class ConstructorTests {

    @Test
    @DisplayName("List manager Constructor should return an Exception with Null
parameter ")
    void listManagerConstructorShouldReturnAnExceptionWithNullParameter() {

        Assertions.assertThrows(NullPointerException.class, () ->new
ListManager(null) );

    }

    @Test
    @DisplayName("list manager constructor should initialize the list attribute
correctly")
    void listManagerConstructorShouldInitializeTheListAttributeCorrectly()
throws Exception {

        List<Element> elements = new ArrayList<Element>();

        Assertions.assertNotNull(new ListManager(elements));

        elements.add(new Element("bananas", 20, 20));

        Assertions.assertNotNull(new ListManager(elements));

    }

}

@Test
@DisplayName("add element to an empty list should add it correctly")
void addElementToAnEmptyListShouldAddItCorrectly() throws Exception {

    List<Element> elements = new ArrayList<>();

    ListManager listManager = new ListManager(elements);

    listManager.addElement( 20,20,"bananas");

    Assertions.assertEquals(1,listManager.returnListSize());

}

@Test
@DisplayName("add element should add correctly a new element to a full list")
void addElementShouldAddCorrectlyANewElementToAFullList() throws Exception {

    List<Element> elements = new ArrayList<>();

    Element element=new Element("bananas",12.0,5);

    elements.add(element);

    element=new Element("strawberries",9.20,4);
```

```

elements.add(element);

element=new Element("apples",10.50,10);

elements.add(element);

ListManager listManager = new ListManager(elements);

listManager.addElement( 20,20,"pineapples");

listManager.addElement(10.02, 30,"pears");

Assertions.assertEquals(5,listManager.returnListSize());

}

@Test
@DisplayName("remove element should return zero if the list is empty or product
not found")
void removeElementShouldReturnZeroIfTheListIsEmptyOrProductNotFound() throws
Exception {

    List<Element> elements = new ArrayList<>();

    ListManager listManager = new ListManager(elements);

    Assertions.assertEquals(0, listManager.removeElement("bananas"));

    Assertions.assertEquals(0, listManager.returnListSize());

    listManager.addElement(20,20,"bananas");

    Assertions.assertEquals(0, listManager.removeElement("strawberries"));

    Assertions.assertEquals(1, listManager.returnListSize());

}

@Test
@DisplayName("remove element from full list with the element should return the
correct number of removed elements")
void removeElementFromFullListShouldReturnTheNumberOfTheRemovedElements() throws
Exception {

    List<Element> elements = new ArrayList<>();

    ListManager listManager = new ListManager(elements);

    listManager.addElement(20,20,"pineapples");

    listManager.addElement(30,20,"strawberries");

    listManager.addElement(30,20,"bananas");

    listManager.addElement(30,20,"bananas");

    Assertions.assertEquals(2,listManager.removeElement("bananas"));

    Assertions.assertEquals(2,listManager.returnListSize());

}

```

```

@Test
@DisplayName("the total amount of a list empty should be zero")
void theTotalAmountOfAListEmptyShouldBeZero() throws Exception {

    List<Element> elements = new ArrayList<>();

    ListManager listManager = new ListManager(elements);

    Assertions.assertEquals(0, listManager.calculateTotalAmount());

}

@Test
@DisplayName("the total amount of a full list should be the correct value")
void theTotalAmountOfAFullListShouldBeTheCorrectValue() throws Exception {

    List<Element> elements = new ArrayList<>();

    ListManager listManager = new ListManager(elements);

    listManager.addElement(20, 20, "pineapples");

    listManager.addElement(30, 20, "strawberries");

    listManager.addElement(30, 20, "bananas");

    Assertions.assertEquals(1600, listManager.calculateTotalAmount());

}

```

1.8. MISSING SOMETHING?

1.9. FAILED TESTS

I test falliti sono per ElementTest: T3, T4.

I test T3 e T4 sono falliti perché nel codice testato questi casi non sono stati presi in considerazione nonostante questi valori inseriti nella stringa productName non dovrebbero essere accettabili.

Il test fallito per ListManager è: T6.

Il test T6 è fallito perché il ciclo creato per rimuovere un elemento dalla lista è sbagliato poiché rimuovendo un elemento in un ciclo che scorre tutta la lunghezza della lista, una volta rimosso, la lunghezza della lista cambia perciò viene sollevata un'eccezione di java.

1.10. CHANGES AFTER FAILED TESTS

Per risolvere il bug rilevato dai test T3 e T4 (Element) abbiamo aggiunto questi due casi come inaccettabili:

```

if (productName == null || productName.equals("") || productName.equals("\n") ||
    productName.equals("\t")) {

    throw new IllegalArgumentException("The product name can't be empty.");

}

```

Per risolvere il bug rilevato dal test T6 (ListManager) abbiamo usato iterator per rimuovere l'elemento dalla lista nel metodo removeElement:

```

public int removeElement(String productName)
{
    int removedElements=0;

    Iterator<Element> iterator=elements.iterator();

    while(iterator.hasNext()){

        if(iterator.next().returnProductName().equals(productName)){
//cambiamento

            iterator.remove();
            removedElements++;

        }

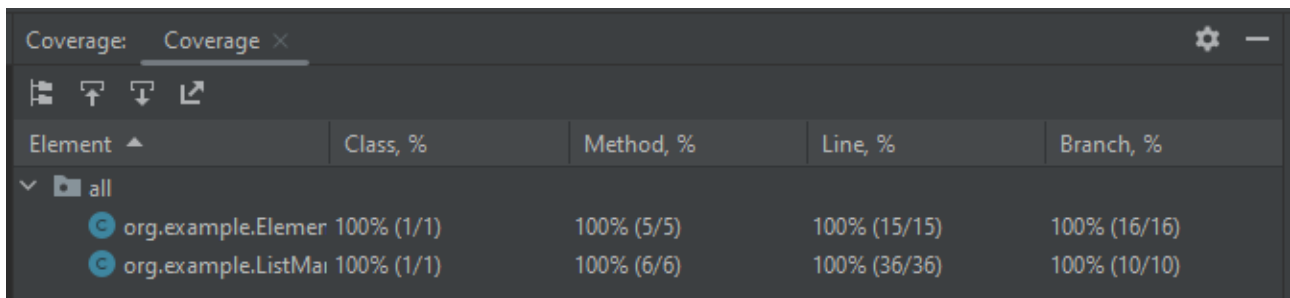
    }

    return removedElements;
}

```

2. HOMEWORK 2

2.1. PRIMA PARTE



Element	Class, %	Method, %	Line, %	Branch, %
all				
org.example.Element	100% (1/1)	100% (5/5)	100% (15/15)	100% (16/16)
org.example.ListMai	100% (1/1)	100% (6/6)	100% (36/36)	100% (10/10)

Dopo aver eseguito il tool di code coverage di IntelliJ abbiamo notato che abbiamo il 100% per la classe Element. Analizzando i vari branch presenti nel codice ci siamo resi conto che per degli statement così semplici la branch coverage è sufficiente. Visto che il 100% di code coverage non implica che la nostra suite di test sia buona abbiamo analizzato nuovamente, in maniera approfondita, il codice ma ci siamo resi conto che i test effettuati erano buoni.

2.2. SECONDA PARTE

2.2.1. CODICE DA TESTARE

```

public double[] calculateMinimums(double[] array){

    double min1;
    double min2;
    double[] output_array = new double[2];

    if(array == null || array.length == 0 || array.length == 1) {

        throw new IllegalArgumentException("the array can't be null, empty or
equals to 1");
    }
}

```



```

    }

    if(array[0]<array[1]){

        min1 = array[0];
        min2 = array[1];

    }
    else {

        min1 = array[1];
        min2 = array[0];

    }
    for(int i = 2; i<array.length; i++) {

        if(array[i]> min1 && array[i] > min2)
            continue;

        if(array[i]< min1) {

            min2=min1;
            min1=array[i];

        }
        else if(array[i] < min2) {

            min2=array[i];

        }

    }

    output_array[0] = min1;
    output_array[1] = min2;

    return output_array;

}

```

2.2.2. TEST PER RAGGIUNGERE IL 100% DI COVERAGE

T1:

```

@Test
@DisplayName("Test for the first branch")
void testFirstBranch() {

    double[] example_array = new double[2];

    example_array[0] = 1;

    example_array[1] = 31;

    Calculate test_obj = new Calculate();

    example_array = test_obj.calculateMinimums(example_array);

    double[] result = new double[2];
}

```

```

    result[0] = 1;

    result[1] = 31;

    Assertions.assertArrayEquals(result, example_array);
}

```

T2:

```

@Test
@DisplayName("Test for the second branch")
void testSecondBranch() {

    double[] example_array = new double[5];

    example_array[0] = 31;

    example_array[1] = 5;

    example_array[2] = 7;

    example_array[3] = 100;

    example_array[4] = 0.1;

    Calculate test_obj = new Calculate();

    example_array = test_obj.calculateMinimums(example_array);

    double[] result = new double[2];

    result[0] = 0.1;

    result[1] = 5;

    Assertions.assertArrayEquals(result, example_array);
}

```

2.2.3. BLACK BOX

Coverage: Coverage ×				
Element ▲	Class, %	Method, %	Line, %	Branch, %
▼ all	33% (2/6)	8% (2/24)	25% (36/142)	27% (22/80)
▼ homework2	100% (2/2)	100% (2/2)	100% (36/36)	91% (22/24)
Calculate	100% (1/1)	100% (1/1)	100% (18/18)	91% (11/12)

In questo caso logicamente ci siamo resi conto che la branch coverage perché l'if statement a rigo 28 non entra mai nel ramo else quindi non potrà mai essere testato questo valore di verità. Per risolvere questo bug bisognerebbe cancellare l'else if e renderlo un else semplice (rigo 34).

Lo statement da così:

```

if(array[i]< min1) {

    min2=min1;
    min1=array[i];

}
else if(array[i] < min2){

    min2=array[i];

}

```

diventa:

```

if(array[i]< min1) {

    min2=min1;
    min1=array[i];

}
else {

    min2=array[i];

}

```

I casi che secondo noi dovrebbero essere testati, basati sulle specifiche del codice.

2.2.3.1.1. UNDESTENDING REQUIREMENTS

Il metodo da testare prende in input un array di doubles e calcola i due valori più piccoli presenti nell'array. Non deve essere possibile calcolarli un array vuoto, nullo o con meno di due elementi. Il metodo fornisce in output un array con due elementi ossia i due valori più piccoli dell'array preso in input.

2.2.3.1.2. EXPLORE WHAT PROGRAM DOES FOR VARIOUS INPUT

```

@Test
@DisplayName("understand inputs of calculate")
void understandInputOfCalculate() {

    double[] array = new double[10];

    for(int i=0; i<10; i+=1){

        array[i]=i;

    }

    Calculate calculate=new Calculate();

    assertNotNull(calculate);

    double[] arrayEqual = new double[2];

    arrayEqual[0]=0;
    arrayEqual[1]=1;

```

```

    assertEquals(arrayEqual, calculate.calculateMinimums(array));
}

```

2.2.3.1.3. EXPLORE INPUTS, OUTPUTS AND IDENTIFY PARTITIONS

- Input:

Double array:

1. Null
2. Empty
3. Lunghezza>=1

- output:

Double array:

1. Null
2. Empty
3. 0<Lunghezza<=2

2.2.3.1.4. IDENTIFY BOUNDARY CASES

Double array:

On point: lunghezza dell'array=2

Off point: lunghezza dell'array=1

2.2.3.1.5. DEVISE TEST CASES

T3: array nullo

T4: array vuoto

T5: array con un elemento

T6: array con due elementi (non eseguito poiché l'on point è stato preso in considerazione nel test 1)

T7: array ordinato in ordine crescente

T8: array ordinato in ordine decrescente

T9: array con lo stesso valore al suo interno

2.2.3.1.6. AUTOMATE TEST CASES

```

@Nested
@DisplayName("writing tests black box")
class BlackBoxTests {

    @Test
    @DisplayName("Array null should return exception")

```

```

void arrayNullShouldReturnException() throws Exception{

    Calculate calculate= new Calculate();

    assertThrows(IllegalArgumentException.class, ()->
calculate.calculateMinimums(null));

}

@Test
@DisplayName("Array empty should return exception")
void arrayEmptyShouldReturnException() throws Exception{

    Calculate calculate= new Calculate();

    double[] array = new double[0];

    assertThrows(IllegalArgumentException.class, ()->
calculate.calculateMinimums(array));

}

@Test
@DisplayName("Array with one element should return exception")
void arrayWithOneElementShouldReturnException() throws Exception{

    Calculate calculate= new Calculate();

    double[] array = new double[1];

    array[0]=1.053;

    assertThrows(IllegalArgumentException.class, ()->
calculate.calculateMinimums(array));

}

@Test
@DisplayName("array in ascendent order")
void arrayInAscendentOrder() {

    double[] array = new double[10];

    for (int i = 0; i < 10; i += 1) {

        array[i] = i;

    }

    Calculate calculate= new Calculate();

    double[] arrayEqual = new double[2];

    arrayEqual[0]=0;
    arrayEqual[1]=1;

    assertArrayEquals(arrayEqual,calculate.calculateMinimums(array));

}

@Test
@DisplayName("array in discendent order")
void arrayInDiscendentOrder() {

```

```

double[] array = new double[10];

int j=100;

for (int i = 0; i < 10; i += 1) {
    array[i] = j;
    j--;
}

Calculate calculate= new Calculate();

double[] arrayEqual = new double[2];

arrayEqual[0]=91;
arrayEqual[1]=92;

assertArrayEquals(arrayEqual,calculate.calculateMinimums(array));
}

@Test
@DisplayName("Array with the same value")
void arrayWithSameValue() {
    double[] array = new double[10];

    for (int i = 0; i < 10; i += 1) {
        array[i] = 5;
    }

    Calculate calculate= new Calculate();

    double[] arrayEqual = new double[2];

    arrayEqual[0]=5;
    arrayEqual[1]=5;

    assertArrayEquals(arrayEqual,calculate.calculateMinimums(array));
}
}

```

2.2.3.1.7. CHANGES AFTER FAILED TESTS

I test T3, T4, T5 sono falliti poiché osservando il metodo ci siamo resi conto che questi tre casi non sono stati gestiti propriamente; perciò, ci siamo occupati di correggere il metodo. Dopo la dichiarazione delle variabili abbiamo inserito un controllo:

```

if(array == null || array.length == 0 || array.length == 1) {
    throw new IllegalArgumentException("the array can't be null, empty or equals to 1");
}

```

3. HOMEWORK 3

3.1. CODICE

Il codice scelto per l'homework 3 è il costruttore della classe Element utilizzata nell'homework 1:

```
public Element(String productName, double cost, int quantity) {  
  
    if (productName == null || productName.equals("") ||  
productName.equals("\n") || productName.equals("\t")) {  
  
        throw new IllegalArgumentException("The product name can't be empty.");  
  
    }  
  
    if(cost < 0.01 || cost > 1000) {  
  
        throw new IllegalArgumentException("The cost can't be negative or equal  
to zero.");  
  
    }  
  
    if(quantity <= 0 || quantity > 1000) {  
  
        throw new IllegalArgumentException("The amount of element can't be  
negative or equal to zero.");  
  
    }  
  
    this.totElement = cost*quantity;  
    this.productName = productName;  
    this.cost = cost;  
    this.quantity = quantity;  
  
}
```

3.2. DEVISE TEST CASE

T1: valori validi di cost e quantity

T2: valori invalidi di quantity

T3: valori invalidi di cost

T4: valori validi di productName

T5: valori invalidi di productName

I test case progettati sono questi poiché ci siamo resi conto che questo metodo ha diverse proprietà tra e quali abbiamo:

- **Variabile cost:**

Divisione partizioni:

- Costo valido => compreso tra 0.01 e 1000 estremi inclusi
- Costo invalido => minore di 0.01 o maggiore di 1000 estremi esclusi

- **Variabile quantity:**

Divisione partizioni:

- Quantità valido => compreso tra 0 e 1000 con 1000 incluso
- Quantità invalido => minore di 0 o maggiore di 1000 con 0 incluso

- **Variabile productName:**

Divisione partizioni:

- Nome prodotto valido => qualsiasi stringa di lunghezza maggiore di zero estremo escluso
- Nome prodotto invalido => stringa nulla o vuota

Per testare questi input abbiamo utilizzato jqwik per generare una serie di valori compresi nel range sia di quelli validi che di quelli invalidi.

3.3. AUTOMATE TEST CASE

```
@Property
@Report(Reporting.GENERATED)
void validValues(@ForAll @IntRange(min = 1, max = 1000) int quantity, @ForAll
@DoubleRange(min = 0.01, max = 1000 ) double cost,
@ForAll("validStringValuesGenerator") String productName) {

    Assertions.assertNotNull(new ElementH3(productName, cost, quantity));
    Statistics.collect(quantity, cost);
}

@Property
@Report(Reporting.GENERATED)
void invalidIntValues(@ForAll("invalidIntValues") int quantity ) {

    Assertions.assertThrows(IllegalArgumentException.class, () -> new
ElementH3("ciao", 20, quantity));
    Statistics.collect(quantity);
}

@Property
@Report(Reporting.GENERATED)
void invalidDoubleValues(@ForAll("invalidDoubleValues") double cost) {

    Assertions.assertThrows(IllegalArgumentException.class, () -> new
ElementH3("ciao", cost, 10));
    Statistics.collect(cost);
}

@Property
@Report(Reporting.GENERATED)
void invalidStringValues(@ForAll("invalidStringValuesGenerator") String
```



```

productName) {

    Assertions.assertThrows(IllegalArgumentException.class, () -> new
    ElementH3(productName, 20, 10));
    Statistics.collect(productName);

}

@property
@Report(Reporting.GENERATED)
void validStringValues(@ForAll("validStringValuesGenerator") String productName)
{

    Assertions.assertNotNull(new ElementH3(productName, 20, 10));
    Statistics.collect(productName);

}

@Provide
private Arbitrary<Integer> invalidIntValues() {

    return Arbitraries.oneOf(Arbitraries.integers().lessOrEqual(0),
    Arbitraries.integers().greaterOrEqual(1001));

}

@Provide
private Arbitrary<Double> invalidDoubleValues() {

    return Arbitraries.oneOf(Arbitraries.doubles().lessThan(0.01),
    Arbitraries.doubles().greaterThan(1000));

}

@Provide
private Arbitrary<String> validStringValuesGenerator() {

    return Arbitraries.strings().ofMinLength(1).ofMaxLength(1000);

}

@Provide
private Arbitrary<String> invalidStringValuesGenerator() {

    return Arbitraries.oneOf(Arbitraries.strings().ofLength(0),
    Arbitraries.just(null));

}

```

3.4. STATISTICHE

Per ogni metodo, è possibile notare, che abbiamo salvato le statistiche con cui sono stati generati certi input. L'abbiamo inserito per tutti i metodi di test poiché volevamo vedere con che frequenza venissero generati i valori più vicini ai boundary delle tre variabili.

```

timestamp = 2023-05-27T10:46:08.801548600, [ElementTestH3:validValues] (1000) statistics =
1 0.01 (6) : 0.60 %
2 0.01 (4) : 0.40 %
1000 0.01 (4) : 0.40 %
999 0.01 (4) : 0.40 %
2 1.0 (3) : 0.30 %
2 1000.0 (3) : 0.30 %
10 0.01 (3) : 0.30 %
92 0.41 (2) : 0.20 %
29 0.09 (2) : 0.20 %
8 3.38 (2) : 0.20 %
37 0.02 (2) : 0.20 %
1000 0.13 (2) : 0.20 %
2 0.07 (2) : 0.20 %
1 1.0 (2) : 0.20 %
1 0.34 (2) : 0.20 %
93 0.01 (2) : 0.20 %
9 0.09 (2) : 0.20 %
1 1000.0 (2) : 0.20 %
1000 1.0 (2) : 0.20 %
1000 1000.0 (2) : 0.20 %
999 1.0 (2) : 0.20 %
2 0.24 (2) : 0.20 %
999 1000.0 (2) : 0.20 %
20 0.02 (1) : 0.10 %
184 8.52 (1) : 0.10 %
7 0.14 (1) : 0.10 %
261 171.75 (1) : 0.10 %
14 0.3 (1) : 0.10 %
162 0.96 (1) : 0.10 %

```

In questo screen significativo dei valori validi di cost e quantity è possibile notare che i valori generati con maggior frequenza sono quelli più vicini al boundary per entrambe le variabili invece tutti gli altri valori non visibili nell'immagine sono stati tutti generati con una frequenza del 0.10%. Questo perché inserendo un range di valori da generare compreso tra 0.01 e 1000 (double) per cost e tra 0 e 1000 per quantity (int) jqwik ha generato tantissimi valori ma con una frequenza minore perché non sono vicini al boundary.

```

timestamp = 2023-05-27T10:46:09.900972700, [ElementTestH3:invalidIntValues] (1000) statistics =
1001 (19) : 1.90 %
-1 (18) : 1.80 %
2147483646 (16) : 1.60 %
-2 (16) : 1.60 %
-2147483648 (16) : 1.60 %
1002 (14) : 1.40 %
0 (14) : 1.40 %
2147483647 (13) : 1.30 %
-2147483647 (12) : 1.20 %
1008 ( 8) : 0.80 %
1003 ( 7) : 0.70 %
-11 ( 6) : 0.60 %
-7 ( 5) : 0.50 %
1006 ( 5) : 0.50 %
-8 ( 5) : 0.50 %
-9 ( 4) : 0.40 %
-6 ( 4) : 0.40 %
1005 ( 4) : 0.40 %
-23 ( 4) : 0.40 %
-16 ( 4) : 0.40 %
-5 ( 4) : 0.40 %
-3 ( 4) : 0.40 %
-22 ( 3) : 0.30 %
1010 ( 3) : 0.30 %
-4 ( 3) : 0.30 %
1013 ( 3) : 0.30 %
1018 ( 3) : 0.30 %
-25 ( 3) : 0.30 %
-20 ( 3) : 0.30 %
1007 ( 3) : 0.30 %
-21 ( 3) : 0.30 %
1023 ( 3) : 0.30 %
-10 ( 3) : 0.30 %
1038 ( 3) : 0.30 %
1061 ( 3) : 0.30 %
-205 ( 2) : 0.20 %
-5250 ( 2) : 0.20 %
1616 ( 2) : 0.20 %
-15 ( 2) : 0.20 %
1050 ( 2) : 0.20 %
-157 ( 2) : 0.20 %
-135 ( 2) : 0.20 %

```

In questo screen significativo è possibile notare che per gli input invalidi di quantity jqwik ha generato in percentuale maggiore i valori vicino al boundary ma in particolare anche valori molto grandi o molto piccoli vicini al valore massimo rappresentabile dall'int in modo tale da esercitare il metodo testato in condizioni "estreme" che potrebbero far uscire fuori un bug.

```

timestamp = 2023-05-27T10:46:10.707855600, [ElementTestH3:invalidStringValue] (1000) statistics =
(259) : 26 %

(257) : 26 %
null (247) : 25 %
(237) : 24 %

```

Per quanto riguarda i valori invalidi di product name jqwik è stato quasi imparziale generando quasi il 25% di ogni input invalido.

[illegible]

Per quanto riguarda i valori validi del nome prodotto j qwik ha generato in percentuale maggiore stringhe di lunghezza 1 perché è il boundary assegnato con frequenza 3,6% e 3,0% e poi con una frequenza del 0.10% stringhe di lunghezza casuale contenenti caratteri casuali. Le stringhe generate con frequenza maggiore sono quelle contenenti uno spazio ed un solo carattere, cosa visibile nell'elenco dei vari valori generati ma non nelle statistiche.

```

timestamp = 2023-05-27T10:46:12.585770800, [ElementTestH3:invalidDoubleValues] (1000) statistics =
1000.01 (24) : 2.40 %
1.7976931348623157E308 (22) : 2.20 %
-1.7976931348623157E308 (20) : 2.00 %
0.0 (19) : 1.90 %
-0.01 (15) : 1.50 %
-1.0 (12) : 1.20 %
1000.08 ( 4) : 0.40 %
-0.08 ( 3) : 0.30 %
-0.03 ( 2) : 0.20 %
-0.59 ( 2) : 0.20 %
-0.07 ( 2) : 0.20 %
1000.02 ( 2) : 0.20 %
-0.27 ( 2) : 0.20 %
-0.02 ( 2) : 0.20 %
1000.61 ( 2) : 0.20 %
1000.6 ( 2) : 0.20 %
1000.1 ( 2) : 0.20 %
-0.12 ( 2) : 0.20 %
1000.15 ( 2) : 0.20 %
1000.63 ( 2) : 0.20 %
-7.674241319021879E204 ( 1) : 0.10 %
1.8907290416154582E24 ( 1) : 0.10 %
405434.08 ( 1) : 0.10 %
-2.5157085338438793E41 ( 1) : 0.10 %
1052.25 ( 1) : 0.10 %

```

Anche per i double, com'è possibile notare i valori generati con più frequenza sono quelli vicini ai boundary o vicini al numero massimo e minimo rappresentabile dal double mentre tutti gli altri sono stati generati con una frequenza del 0.10% per gli stessi motivi già discussi in precedenza.

4. RIFERIMENTI

Il report con le statistiche di jqwik è possibile consultarlo al file: [report jqwik.html](#)

I report di coverage per i vari homework è possibile consultarli nella cartella coverage.

Per l'homework3 abbiamo eseguito ugualmente il tool di coverage anche se non richiesto dalla traccia e anche se non discusso nella documentazione. Lo abbiamo inserito poiché l'abbiamo eseguito per vedere se avevamo coperto tutte le partizioni.