

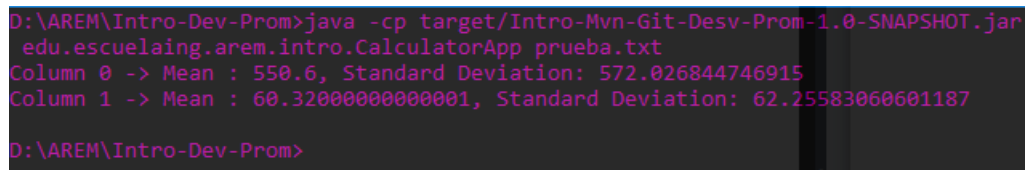
# Media y Desviación Estándar (Introducción Maven Git y GitHub )

Julián Eduardo Arias Barrera

Agosto 2020

## 1 Introducción

Es una practica de utilización de Maven y Git con un programa que puede calcular la media y desviación estándar de un conjunto de datos.



```
D:\AREM\Intro-Dev-Prom>java -cp target/Intro-Mvn-Git-Desv-Prom-1.0-SNAPSHOT.jar
edu.escuelaing.arem.intro.CalculatorApp prueba.txt
Column 0 -> Mean : 550.6, Standard Deviation: 572.026844746915
Column 1 -> Mean : 60.32000000000001, Standard Deviation: 62.25583060601187
D:\AREM\Intro-Dev-Prom>
```

Figure 1: Resultados de dos Columnas de Datos

Dado que la media es :

$$\frac{\sum_{i=1}^n x_i}{n}$$

Y la desviación estándar es :

$$\sqrt{\frac{\sum_{i=1}^n (x_i - x_{avg})^2}{n - 1}}$$

Se creo una implementación propia de LinkedList y se utilizo interfaces funcionales dando la posibilidad de ampliar las funciones de la calculadora escrita.

## 2 Instalación y Ejecución

Con la terminal debemos ejecutar los siguientes comandos:

- Debemos clonar el repositorio con la instrucción:

```
> git clone https://github.com/AriasAEnima/Intro-MVN-GIT-Desviacion-Promedio.git
```

- Compilar con Maven :

```
... /Intro-MVN-GIT-Desviacion-Promedio> mvn package
```

- Finalmente correr la calculadora con los datos deseados , en este caso tomara dos columnas de datos puestas en un archivo txt que está en el directorio raíz:

```
>java -cp target/Intro-Mvn-Git-Desv-Prom-1.0-SNAPSHOT.jar
edu.escuelaing.arem.intro.CalculatorApp prueba.txt
```

Nos mostrara los resultados:

```
D:\AREM\Intro-Dev-Prom>java -cp target/Intro-Mvn-Git-Desv-Prom-1.0-SNAPSHOT.jar
edu.escuelaing.arem.intro.CalculatorApp prueba.txt
Column 0 -> Mean : 550.6, Standard Deviation: 572.026844746915
Column 1 -> Mean : 60.32000000000001, Standard Deviation: 62.25583060601187
D:\AREM\Intro-Dev-Prom>
```

Figure 2: Resultados de dos Columnas de Datos

### 3 Diseño

Las interfaces definidas como DoubleMath/IntegerMath serian una Interfaz Funcional (cada una) y los creados por medio de notación lambda como clases anónimas: adición, sustracción, media y desviación estándar. Podríamos decir que un acercamiento a un patrón de comando en el cual y el método "operation(...)" es un símil de "execute()" del patrón; como dije es un acercamiento ya que utilizamos interfaces funcionales para emplear contenedores genéricos. [1]

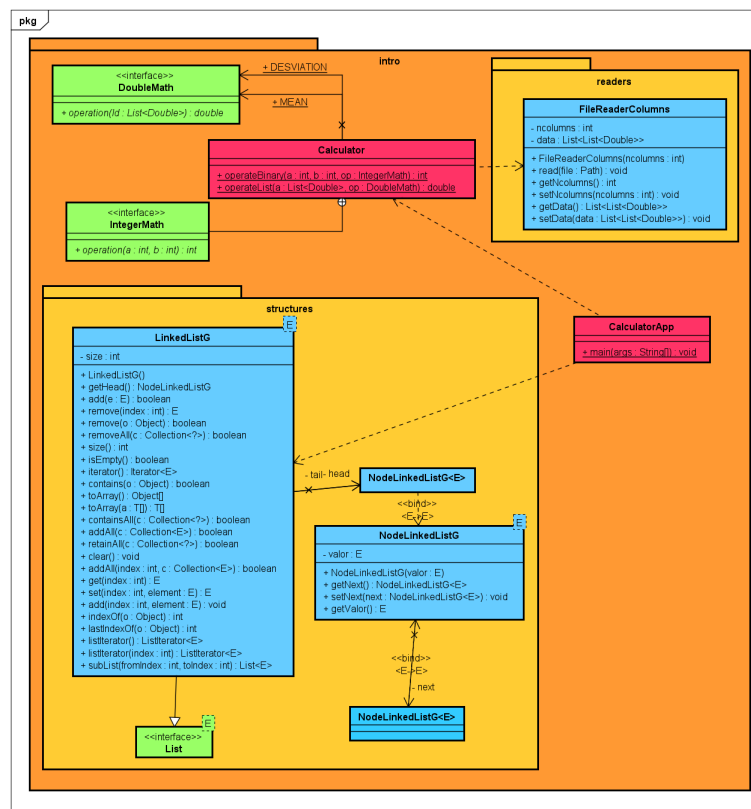


Figure 3: Modelo

También se utilizó el patrón Iterator en la implementación de LinkedListG (nótese la diferencia en la G) usando una implementación propia de Nodos e Iteradores de Lista. (Iterator es superinterfaz de ListIterator). [2]

Estas dos implementaciones nos permite agregar funciones fácilmente y ejecutarlas utilizando contenedores que permiten utilizar cualquier tipo de objeto, y que el compilador nos ayudara a garantizar que estos serán de un mismo tipo. Pues podremos pasarle la operación a la función si no es ninguna de las 4 que ya tenemos definidas.

También podemos ver partes del código inicialmente solo se leerá dos columnas de datos pero esto podrá ser cambiado implementando mas funciones o otros Readers:

```
Path file = Paths.get(args[0]);
FileReaderColumns frc=new FileReaderColumns(2);
frc.read(file);
List<List<Double>> data=frc.getData();
```

Figure 4: Lector de Archivos en la clase Calculator App

Algunas clases anónimas definidas como variables finales, aunque estas dos se hayan definido no significa que no pueden haber mas, y que deben ser definidas dentro de calculadora, de hecho pueden ser directamente enviadas a los métodos (lo veremos en las pruebas):

```
public class Calculator {

    public static final Calculator.DoubleMath MEAN=(a)->{
        Double ans=0.0;
        for(Double n:a){
            ans+=n;
        }
        return ans/a.size();
    };

    public static final Calculator.DoubleMath DESVIATION=(a)->{
        Double m=Calculator.operateList(a, MEAN);
        Double ans=0.0;
        for(Double n:a){
            ans+=Math.pow((n-m),2);
        }
        return Math.sqrt(ans/(a.size()-1));
    };

    public interface IntegerMath {
        int operation(int a, int b);
    }

    /**
     * Permite hacer operaciones sobre una lista de double's.
     */
    public interface DoubleMath {
        double operation(List<Double> ld);
    }

    public static int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    }

    /**
     * Ejecuta una operacion sobre una lista de doubles.
     * @param a la lista
     * @param op la operacion
     * @return resultado de la operacion
     */
    public static double operateList(List<Double> a, DoubleMath op) {
        return op.operation(a);
    }
}

public class LinkedListG<E> implements List<E>{
    private NodeLinkedListG<E> head;
    private NodeLinkedListG<E> tail;
    private int size;

    /**
     * Crea la lista con cabeza y cola en nulo
     */
    public LinkedListG() {
        head=null;
        tail=null;
        size=0;
    }

    /**
     * @return devuelve la cabeza del linked list
     */
    public NodeLinkedListG getHead(){
        return head;
    }

    /**
     * Con la estrategia de Linked List agrega elementos al final
     * @param e el valor del nodo
     */
    @Override
    public boolean add(E e) {
        NodeLinkedListG<E> nuevo=new NodeLinkedListG(e);
        if (head==null){
            tail= head= nuevo;
        }else{
            tail.setNext(nuevo);
            tail=nuevo;
        }
        size++;
        return true;
    }
}
```

(a) Clase Calculadora

(b) Clase LinkedListG

Figure 5: Clases

## 4 Pruebas

Aquí se probó que se permitiera utilizar operaciones no definidas en calculadora:

```
private final Calculator.IntegerMath ADDITION = (a, b) -> a + b;
private final Calculator.IntegerMath SUBTRACTION = (a, b) -> a - b;

/**
 * Sets up the test fixture.
 * (Called before every test case method.)
 */

@Test
public void enteros(){
    try{
        System.out.println("40 + 2 = " +
            Calculator.operateBinary(40, 2, ADDITION));
        System.out.println("20 - 10 = " +
            Calculator.operateBinary(20, 10, SUBTRACTION));
        assertTrue(true);
    }catch(Exception e){
        fail("Error");
    }
}
```

Figure 6: Una prueba con operaciones definidas por fuera de la clase calculadora

Estilo de pruebas con datos para el calculo de media y desviación estándar:

```
@Test
public void Datos1() {
    List<Double> lista=new LinkedList<Double>();
    lista.add(160.0);
    lista.add(591.0);
    lista.add(114.0);
    lista.add(229.0);
    lista.add(230.0);
    lista.add(270.0);
    lista.add(128.0);
    lista.add(1657.0);
    lista.add(624.0);
    lista.add(1503.0);
    Double ans1m=Calculator.operateList(lista, MEAN);
    Double ans2m=Calculator.operateList(lista, DESVIATION);
    assertEquals(550.6, ans1m,0.0001);
    assertEquals(572.03, ans2m,0.005);
}
```

Figure 7: Pruebas con datos

Se hicieron en total 4 pruebas:

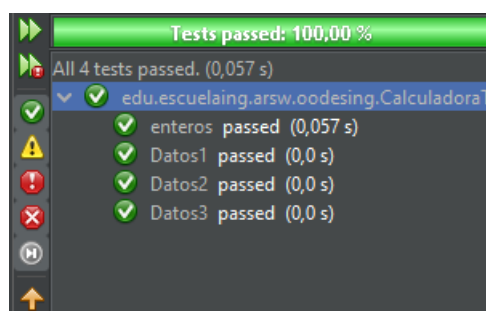


Figure 8: Pruebas con datos

## 5 Conclusiones

La ventaja de los contendores genéricos como lo fue la implementación de LinkedList y las interfaces funcionales de la calculadora es que disminuye notablemente el acoplamiento .

## References

- [1] Refactoring. *Command*. URL: <https://refactoring.guru/design-patterns/command>. (accessed: 12.08.2020).
- [2] Refactoring. *Iterator*. URL: <https://refactoring.guru/design-patterns/iterator>. (accessed: 12.08.2020).