Semestre I – 2021

Algoritmos y
estructuras de datos

Integrantes:

Jose Mariano Reyes

Andrés de la Roca

Investigación – Singleton

Este patrón de diseño se encarga de restringir la creación de objetos, y se asegura de que una clase solo tenga una única instancia y proporciona un acceso global a la misma.

Ventajas:

- Reduce el espacio de nombres: Este patrón prácticamente es una mejora sobre las variables globales, ya que no se reservan nombres para las diferentes variables globales que puedan existir sino que ahora se maneja por medio de instancias.
- Controla el acceso a la instancia única: La clase de Singleton encapsula a una única instancia, de esta manera se puede controlar como y cuando se accede a esta instancia.
- Patrón flexible: Es un patrón bastante flexible en cuanto al numero variable de instancias ya que se le puede configurar como para que se permite mas de una solo instancia.

Desventajas:

- Operación síncrona: Este patrón en su forma básica funciona de manera síncrona por lo cual si nuestro programa requiere de llamar al método singleton muchas veces podría ser perjudicial para el rendimiento.
- Recurso compartido: Si no se maneja correctamente el estado de la instancia este se puede ver alterado por partes de nuestro programa y podrían llegar a afectar el funcionamiento de todo el programa a causa de esto.

¿El uso de este patrón de diseño es el adecuado en este programa?:

Este patrón si es adecuado para nuestro programa, ya que nos ayuda a asegurarnos de que solo se llame a una sola instancia de la clase calculadora, de esta manera nos aseguramos de que solo se llame a solo una referencia de la clase. Su comportamiento como una variable global mejorada le encaja muy bien a lo que se quiere alcanzar con este programa.

Pruebas Unitarias - JUnit

Integrantes:

Jose Mariano Reyes

Andrés de la Roca

- **Pruebas unitarias clase de calculadora**

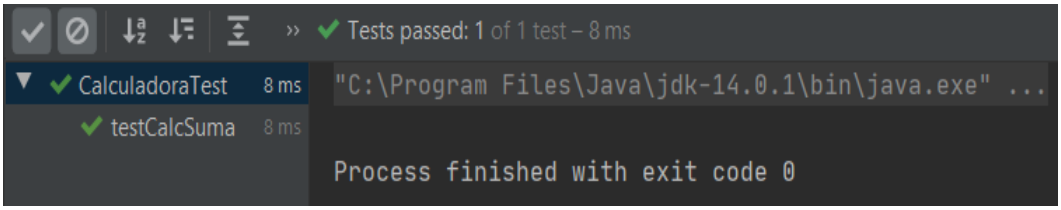
1. Test de suma

```
@Test
public void testCalcSuma() {
    assertEquals("3", calc.Calculo("2 1 +"));
}
```

Para esta prueba se quiere comprobar el correcto funcionamiento de la implementación de suma a la calculadora

En esta prueba se suma 1 + 2, como resultado se espera un 3.

La prueba fue completada exitosamente.

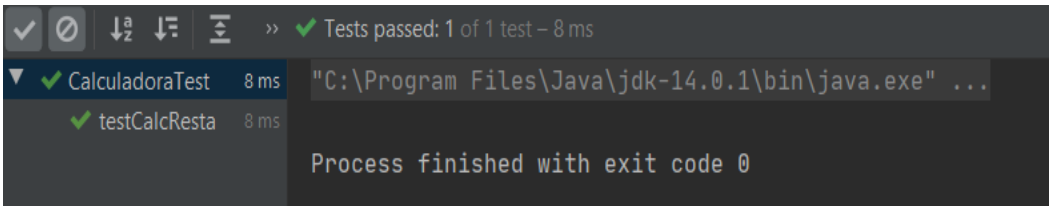


2. Test de resta

```
@Test
public void testCalcResta(){
    assertEquals("4", calc.Calculo("4 8 -"));
}
```

Para esta prueba se quiere comprobar el correcto funcionamiento de la implementación de resta a la calculadora.

En esta prueba se resta 8 – 4, como resultado se espera un 4.
La prueba fue completada exitosamente



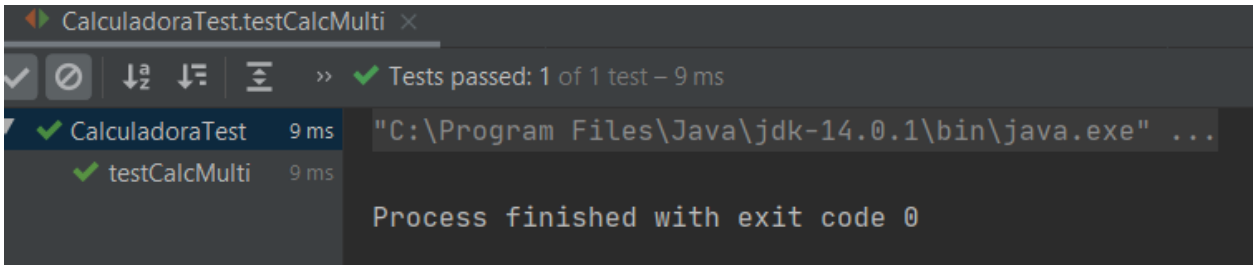
3. Test de multiplicación

```
@Test
public void testCalcMulti() {
    assertEquals("12", calc.Calculo("4 3 *"));
}
```

Para esta prueba se quiere comprobar la correcta implementación de la multiplicación en la calculadora

En esta prueba se multiplica 4 * 3 y se espera un resultado de 12.

La prueba fue completada exitosamente



4. Test de división

```
@Test
public void testCalcDivision(){
    assertEquals("2", calc.Calculo("1 2 /"));
}
```

Para esta prueba se quiere comprobar el correcto funcionamiento de la implementación de división en la calculadora.

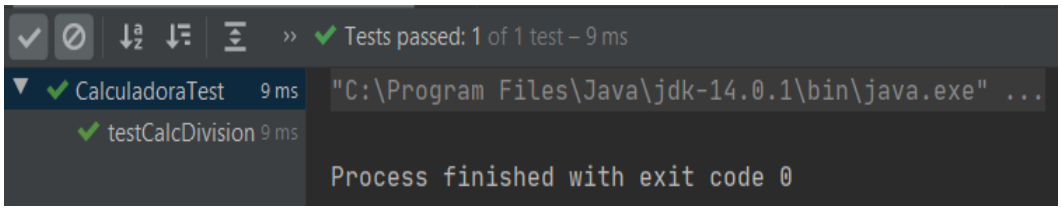
En esta prueba se divide 2/1 y se espera un resultado de 2.

La prueba fue completada exitosamente.

Integrantes:

Jose Mariano Reyes

Andrés de la Roca



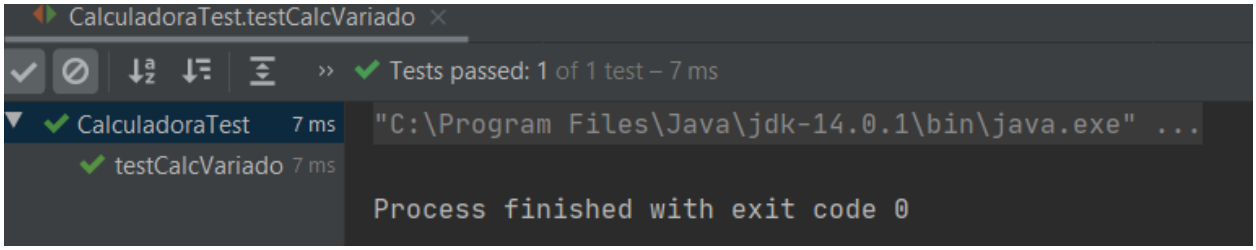
5. Test variado

```
@Test
public void testCalcVariado() {
    assertEquals("15", calc.Calculo("1 2 + 4 * 3 +"));
}
```

Para esta prueba se quiere comprobar el correcto funcionamiento de la implementación de varias operaciones dentro de la calculadora

En esta prueba se ingresa la operación (((1+2)*4)+3) y se espera como resultado 15.

La prueba fue completada exitosamente



- **Pruebas unitarias clase de pilas**

1. Test de push y pop

```
@Test
public void TestPushPop() {
    stack.push("Prueba1");
    assertEquals("Prueba1", stack.pop());
}
```

Integrantes:

Jose Mariano Reyes

Andrés de la Roca

Para este test se quiere probar la funcionalidad de almacenamiento del stack y como nos puede mostrar el contenido del ultimo objeto ingresado.

2. Test de peek

```
@Test
public void TestPeek(){
    stack.push("Touhou");
    assertEquals("Touhou", stack.peek());
}
```

Para este test se quiere probar la funcionalidad de que el stack devuelva el ultimo objeto ingresado.

3. Test de size

```
@Test
public void Testsize() {
    stack.push("CR7");
    stack.push("Binario");
    stack.push("Avicii");
    stack.push("Jeepeta");
    assertEquals(4, stack.size());
}
```

Para esta prueba se lleno de 4 objetos el stack y luego se probo la función size para comprobar si devolvía la cantidad correcta de objetos en el stack.

4. Test de empty

```
@Test public void TestEmpty() {
    assertEquals(true, stack.empty());
    stack.push("Hola");
    assertEquals(false, stack.empty());
}
```

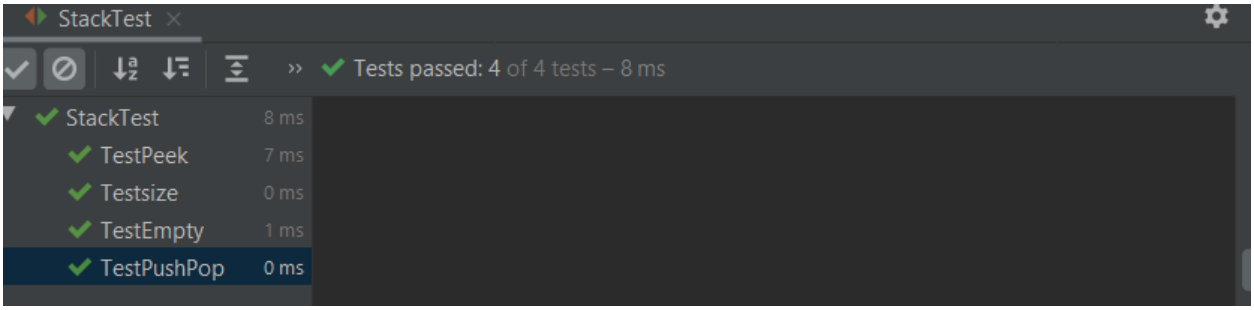
Para esta prueba se comprueba que este vacio el stack, para luego meterle un objeto y comprobar si el boolean de la función empty cambia de estado al meter un objeto

Integrantes:

Jose Mariano Reyes

Andrés de la Roca

Las pruebas fueron superadas con éxito



- **Pruebas unitarias clase de listas encadenadas (dobles y simples)**

1. Push y Pop de lista encadenada simple

```
@Test
public void TestPushPopSingle() {
    singlelinked.push("Prueba1");
    assertEquals("Prueba1", singlelinked.pop());
}
```

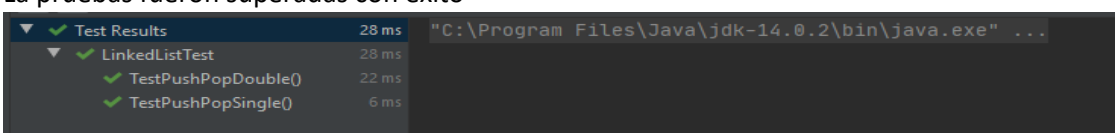
Con este test nos aseguramos que la lista simplemente encadenada pueda pushear un elemento para luego popearlo

2. Push y Pop de lista encadenada doble

```
@Test
public void TestPushPopDouble(){
    doublelinked.push("Touhou");
    assertEquals("Touhou", doublelinked.pop());
}
```

Con este test nos aseguramos que la lista doblemente encadenada pueda pushear un elemento para luego popearlo

La pruebas fueron superadas con éxito



Bibliografía

"Patrón de Diseño Singleton (Creación)." *Informaticapc.com*, 2010, informaticapc.com/patrones-de-diseno/singleton.php

"Singleton." *Reactiveprogramming.io*, reactiveprogramming.io/blog/es/patrones-de-diseno/singleton