

Práctica Clean Code Parte 2

Para la segunda parte he mantenido mi programa anterior pero haciendo algunos cambios que me ayuden en la demostración de este segundo trabajo. Lo más destacable es que he añadido tres clases y otras funciones.

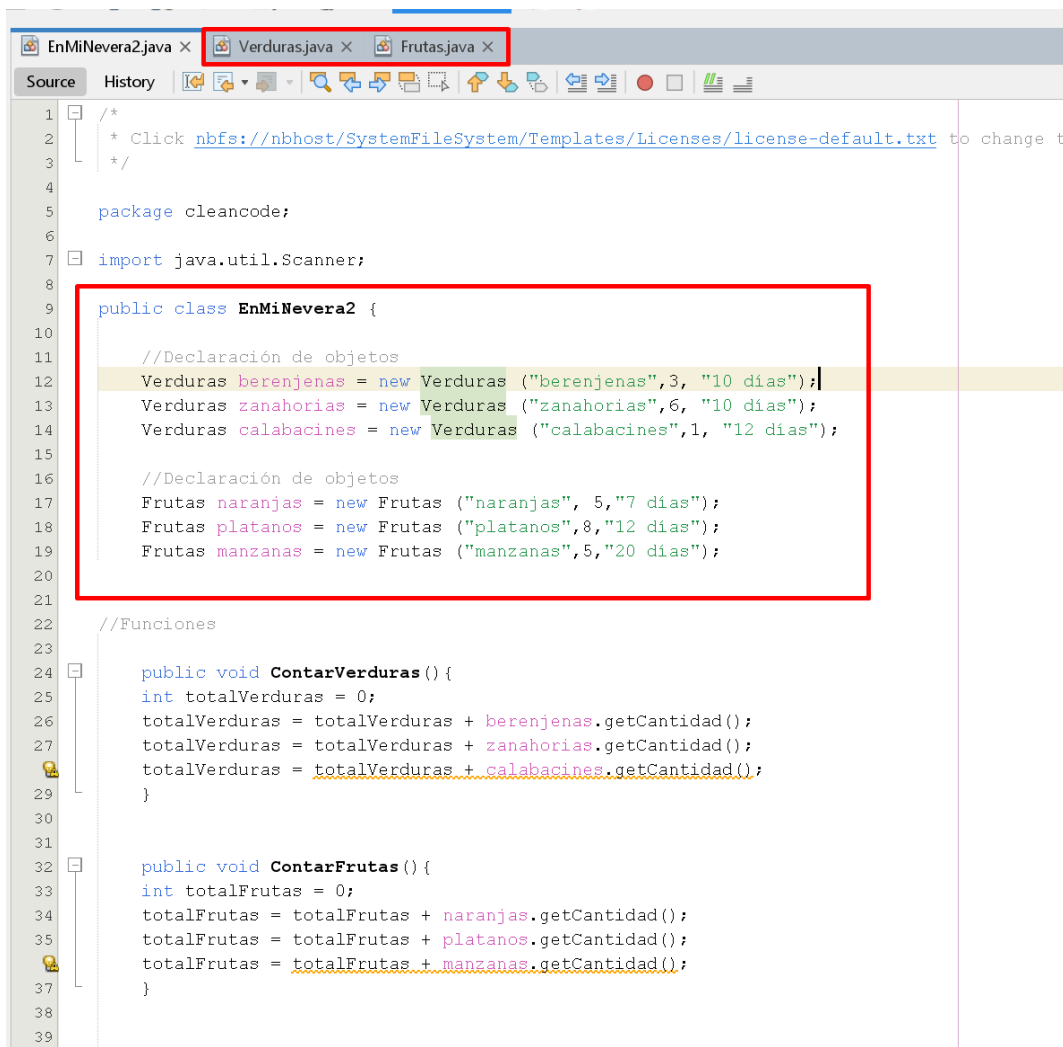
Este trabajo está realizado con JDK 21 y este es el enlace GitHub

<https://github.com/taniagarciaDAM/CleanCode>

- Bloque 4: OBJETOS Y ESTRUCTURAS DE DATOS

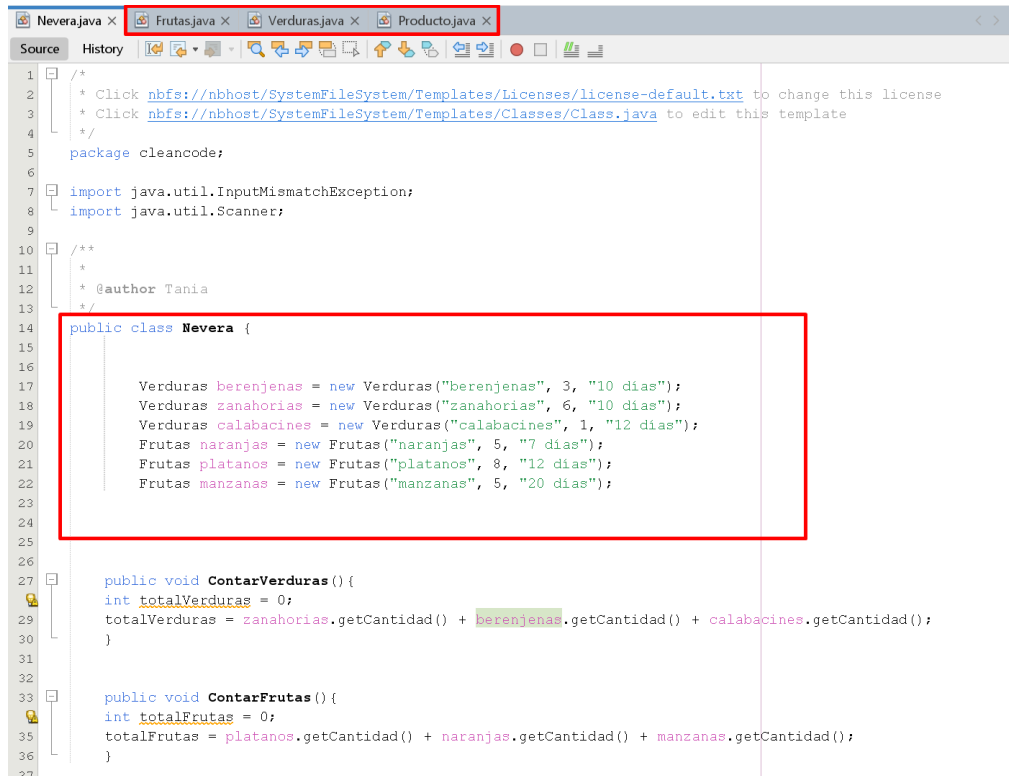
Para aplicar este bloque podemos tener en cuenta:

1. **Reducción de Acoplamientos:** Evita que un módulo conozca los detalles internos de los objetos que manipula. Podemos evitar que la clase EnMiNevera2 acceda directamente a las propiedades internas de los objetos Verduras y Frutas.
2. **Uso de Métodos de Interfaz:** En lugar de acceder directamente a las propiedades internas de los objetos, proporcionamos métodos de interfaz que expongan solo la funcionalidad necesaria para que otros módulos interactúen con esos objetos.
3. **Encapsulamiento:** La implementación interna de un objeto permanece oculta para los consumidores externos. Esto significa que los detalles internos de la clase no deben ser accesibles desde fuera de la clase.



```
1  /*
2   * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change t
3   */
4
5  package cleancode;
6
7  import java.util.Scanner;
8
9  public class EnMiNevera2 {
10
11      //Declaración de objetos
12      Verduras berenjenas = new Verduras ("berenjenas",3, "10 días");
13      Verduras zanahorias = new Verduras ("zanahorias",6, "10 días");
14      Verduras calabacines = new Verduras ("calabacines",1, "12 días");
15
16      //Declaración de objetos
17      Frutas naranjas = new Frutas ("naranjas", 5,"7 días");
18      Frutas platanos = new Frutas ("platanos",8,"12 días");
19      Frutas manzanas = new Frutas ("manzanas",5,"20 días");
20
21
22      //Funciones
23
24      public void ContarVerduras() {
25          int totalVerduras = 0;
26          totalVerduras = totalVerduras + berenjenas.getCantidad();
27          totalVerduras = totalVerduras + zanahorias.getCantidad();
28          totalVerduras = totalVerduras + calabacines.getCantidad();
29      }
30
31
32      public void ContarFrutas() {
33          int totalFrutas = 0;
34          totalFrutas = totalFrutas + naranjas.getCantidad();
35          totalFrutas = totalFrutas + platanos.getCantidad();
36          totalFrutas = totalFrutas + manzanas.getCantidad();
37      }
38
39  }
```

Los objetos de Verduras y Frutas se instancian dentro de la clase EnMiNevera2, y luego se accede a ellos directamente desde los métodos de esta clase. Sin embargo, la manipulación de los objetos Verduras y Frutas se hace a través de métodos internos definidos en la clase Producto, lo que reduce el acoplamiento entre EnMiNevera2 y los detalles internos de los objetos Verduras y Frutas. Así, no expone la estructura interna de los objetos a través de la encapsulación y la comunicación indirecta.



```
1  /*
2   * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
3   * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
4   */
5   package cleancode;
6
7   import java.util.InputMismatchException;
8   import java.util.Scanner;
9
10  /**
11   *
12   * @author Tania
13   */
14  public class Nevera {
15
16
17      Verduras berenjenas = new Verduras("berenjenas", 3, "10 dias");
18      Verduras zanahorias = new Verduras("zanahorias", 6, "10 dias");
19      Verduras calabacines = new Verduras("calabacines", 1, "12 dias");
20      Frutas naranjas = new Frutas("naranjas", 5, "7 dias");
21      Frutas platanos = new Frutas("platanos", 8, "12 dias");
22      Frutas manzanas = new Frutas("manzanas", 5, "20 dias");
23
24
25
26
27  public void ContarVerduras() {
28      int totalVerduras = 0;
29      totalVerduras = zanahorias.getCantidad() + berenjenas.getCantidad() + calabacines.getCantidad();
30  }
31
32
33  public void ContarFrutas() {
34      int totalFrutas = 0;
35      totalFrutas = platanos.getCantidad() + naranjas.getCantidad() + manzanas.getCantidad();
36  }
37  }
```

```
1  /*
2  * Click nbfs://nbhost/SystemFileSystem/Templates/Licenses/license-default.txt to change this license
3  * Click nbfs://nbhost/SystemFileSystem/Templates/Classes/Class.java to edit this template
4  */
5  package cleancode;
6
7  /**
8   *
9   * @author Tania
10  */
11  public class Producto {
12
13      private String nombre;
14      private int cantidad;
15      private final String CADUCIDAD;
16
17      public Producto(String nombre, int cantidad, final String CADUCIDAD) {
18          if (nombre == null || nombre.trim().isEmpty()) {
19              throw new IllegalArgumentException("El nombre del producto no puede estar vacío");
20          }
21          this.nombre = nombre;
22          this.cantidad = cantidad;
23          this.CADUCIDAD = CADUCIDAD;
24      }
25
26      public int getCantidad() {
27          return cantidad;
28      }
29
30      public void setCantidad(int cantidad) {
31          this.cantidad = cantidad;
32      }
33
34      public String getNombre() {
35          return nombre;
36      }
37
38      public void setNombre(String nombre) {
39          this.nombre = nombre;
40      }
41  }
```

- Bloque 5: MANEJO DE ERRORES

Con estos cambios en la estructura, podemos ver además otra herramienta importante dentro de Clean Code: las excepciones y el manejo de datos nulos.

En esta función vamos a lanzar una excepción en caso de error al añadir un producto que no esté en la lista o una cantidad no válida.

```

public void agregarProducto(Scanner scanner, String producto) throws InputMismatchException {
    System.out.println("Ingrese la cantidad de " + producto + " que ha comprado:");
    try {
        int cantidadComprada = scanner.nextInt();
        switch (producto) {
            case "naranjas":
                naranjas.agregarCantidad(cantidadComprada);
                break;
            case "platanos":
                platanos.agregarCantidad(cantidadComprada);
                break;
            case "manzanas":
                manzanas.agregarCantidad(cantidadComprada);
                break;
            case "berenjenas":
                berenjenas.agregarCantidad(cantidadComprada);
                break;
            case "zanahorias":
                zanahorias.agregarCantidad(cantidadComprada);
                break;
            case "calabacines":
                calabacines.agregarCantidad(cantidadComprada);
                break;
            default:
                throw new InputMismatchException("Producto no válido");
        }
    } catch (InputMismatchException e) {
        System.out.println("Error: Se esperaba un número entero para la cantidad");
    } finally {
        scanner.nextLine(); // Limpiar el buffer del scanner
    }
}

```

Además utilizamos try-catch-finally para asegurarnos de que se manejan adecuadamente las excepciones y se deja el programa en un estado consistente.

Ahora nos centramos en la clase Producto, en este caso modificamos el constructor para asegurarnos de que no haya un dato nulo.

```

public Producto(String nombre, int cantidad, final String CADUCIDAD) {
    if (nombre == null || nombre.trim().isEmpty()) {
        throw new IllegalArgumentException("El nombre del producto no puede estar vacío");
    }
    this.nombre = nombre;
    this.cantidad = cantidad;
    this.CADUCIDAD = CADUCIDAD;
}

```

- Bloque 6: TESTS UNITARIOS

Hemos hecho algunos test para probar que el código funciona bien.

```

@BeforeEach
public void setUp() {
    Verduras berenjenas = new Verduras("berenjenas", 3, "10 días");
    Verduras zanahorias = new Verduras("zanahorias", 6, "10 días");
    Verduras calabacines = new Verduras("calabacines", 1, "12 días");
    Frutas naranjas = new Frutas("naranjas", 5, "7 días");
    Frutas platanos = new Frutas("platanos", 8, "12 días");
    Frutas manzanas = new Frutas("manzanas", 5, "20 días");
}

@Test
public void testContarVerduras() {

    int totalVerduras = zanahorias.getCantidad() + berenjenas.getCantidad() + calabacines.getCantidad();

    // Se verifica que el resultado sea el esperado
    assertEquals(10, totalVerduras);
}

@Test
public void testContarFrutas() {

    int totalFrutas = platanos.getCantidad() + naranjas.getCantidad() + manzanas.getCantidad();

    assertEquals(18, totalFrutas); // Se espera 18 frutas en total
}

@Test
public void testAgregarProducto() {
    // Prueba agregar un número válido de naranjas
    Scanner scanner1 = new Scanner("5");
    assertDoesNotThrow(() -> nevera.agregarProducto(scanner1, "naranjas"));
    assertEquals(10, nevera.naranjas.getCantidad()); // Se espera que la cantidad de naranjas sea 10

    // Prueba agregar un número inválido de plátanos
    Scanner scanner2 = new Scanner("cincuenta");
    assertThrows(InputMismatchException.class, () -> nevera.agregarProducto(scanner2, "platanos")); // Se espera una
    assertEquals(8, nevera.platanos.getCantidad()); // Se espera que la cantidad de plátanos siga siendo 8

    // Prueba agregar un número negativo de manzanas
    Scanner scanner3 = new Scanner("-3");
    assertThrows(IllegalArgumentException.class, () -> nevera.agregarProducto(scanner3, "manzanas")); // Se espera un
    assertEquals(5, nevera.manzanas.getCantidad()); // Se espera que la cantidad de manzanas siga siendo 5
}

```

Las pruebas **testContarVerduras** y **testContarFrutas** se centran en una sola comprobación, evaluando la cantidad total de verduras y frutas respectivamente. En general, las pruebas tienden a abordar un solo concepto, como se evidencia en estas dos pruebas.

Las pruebas siguen los principios de la regla F.I.R.S.T. Son rápidas, independientes, repetibles y auto-validables, lo que las hace adecuadas para su ejecución en diferentes entornos y asegura su fiabilidad. Además, son oportunas, ya que están escritas antes del código de producción para probar su comportamiento esperado.

En cuanto a la limpieza de las pruebas, estas están diseñadas con legibilidad en mente, sirviendo como documentación clara del comportamiento esperado del código. Por último, se sigue la regla de "Un Assert por test", lo que garantiza que cada prueba se enfoque en un solo aspecto del comportamiento, manteniendo la simplicidad y claridad.

- Bloque 7: CLASES

Mis clases:

1.Producto

```

L  */
  package cleancode;

  /**
   *
   * @author Tania
   */
  public class Producto {

      private String nombre;
      private int cantidad;
      private final String CADUCIDAD;

      public Producto(String nombre, int cantidad, final String CADUCIDAD) {
          if (nombre == null || nombre.trim().isEmpty()) {
              throw new IllegalArgumentException("El nombre del producto no puede estar vacío");
          }
          this.nombre = nombre;
          this.cantidad = cantidad;
          this.CADUCIDAD = CADUCIDAD;
      }

      public int getCantidad() {
          return cantidad;
      }

      public void setCantidad(int cantidad) {
          this.cantidad = cantidad;
      }

      public String getNombre() {
          return nombre;
      }

      public void setNombre(String nombre) {
          this.nombre = nombre;
      }

      // Método común para agregar cantidad a un producto
      public void agregarCantidad(int cantidad) {
          this.cantidad += cantidad;
      }
  }

```

2. Verduras

```

  /**
   *
   * @author Tania
   */
  public class Verduras extends Producto {
      public Verduras(String nombre, int cantidad, final String CADUCIDAD) {
          super(nombre, cantidad, CADUCIDAD);
      }
  }

```

3. Frutas

```

  * @author Tania
  */
  public class Frutas extends Producto {
      public Frutas(String nombre, int cantidad, final String CADUCIDAD) {
          super(nombre, cantidad, CADUCIDAD);
      }
  }

```

4. Nevera

```

public class Nevera {

    Verduras berenjenas = new Verduras("berenjenas", 3, "10 días");
    Verduras zanahorias = new Verduras("zanahorias", 6, "10 días");
    Verduras calabacines = new Verduras("calabacines", 1, "12 días");
    Frutas naranjas = new Frutas("naranjas", 5, "7 días");
    Frutas platanos = new Frutas("platanos", 8, "12 días");
    Frutas manzanas = new Frutas("manzanas", 5, "20 días");

    public void ContarVerduras() {
        int totalVerduras = 0;
        totalVerduras = zanahorias.getCantidad() + berenjenas.getCantidad() + calabacines.getCantidad();
    }

    public void ContarFrutas() {
        int totalFrutas = 0;
        totalFrutas = platanos.getCantidad() + naranjas.getCantidad() + manzanas.getCantidad();
    }

    public void agregarProducto(Scanner scanner, String producto) throws InputMismatchException {
        System.out.println("Ingrese la cantidad de " + producto + " que ha comprado:");
        try {
            int cantidadComprada = scanner.nextInt();
            switch (producto) {
                case "naranjas":
                    naranjas.agregarCantidad(cantidadComprada);
                    break;
                case "platanos":
                    platanos.agregarCantidad(cantidadComprada);
                    break;
                case "manzanas":
                    manzanas.agregarCantidad(cantidadComprada);
                    break;
                case "berenjenas":
                    berenjenas.agregarCantidad(cantidadComprada);
                    break;
                case "zanahorias":
                    zanahorias.agregarCantidad(cantidadComprada);
                    break;
                case "calabacines":
                    calabacines.agregarCantidad(cantidadComprada);
                    break;
                default:
                    throw new InputMismatchException("Producto no válido");
            }
        } catch (InputMismatchException e) {
            System.out.println("Error: Se esperaba un número entero para la cantidad");
        } finally {
            scanner.nextLine(); // Limpiar el buffer del scanner
        }
    }

    public void mostrarInventario() {
        System.out.println("Inventario en la nevera:");
        System.out.println("Naranjas: " + naranjas.getCantidad());
        System.out.println("Plátanos: " + platanos.getCantidad());
        System.out.println("Manzanas: " + manzanas.getCantidad());
        System.out.println("Berenjenas: " + berenjenas.getCantidad());
        System.out.println("Zanahorias: " + zanahorias.getCantidad());
        System.out.println("Calabacines: " + calabacines.getCantidad());
    }
}

```

Las clases Verduras y Frutas están organizadas siguiendo una estructura similar a la de la clase Producto, con constantes, variables y métodos adecuadamente distribuidos. Heredan de la clase Producto, lo que implica que comparten propiedades y comportamientos comunes relacionados con los productos, como el nombre, la cantidad y la caducidad. Este enfoque de herencia permite una reutilización efectiva del código y promueve la cohesión, ya que las

clases relacionadas están agrupadas lógicamente bajo una jerarquía común. Además, están centradas en su objetivo principal y no deberían crecer indiscriminadamente.

Las clases están diseñadas para ser modificables y extensibles. Esto implica que la modificación de una clase no debería afectar significativamente a otras partes del sistema, siguiendo así el Principio Open-Closed y el Principio de Inversión de Dependencias y reduciendo la posibilidad de errores y comportamientos inesperados.

En resumen, las clases Verduras y Frutas, al extender la clase Producto, ofrecen un enfoque cohesivo, modular y seguro para representar y manipular productos dentro del sistema, lo que contribuye a un diseño robusto y mantenible en el desarrollo de software.