EJERCICIO TEMA 6
ALMACEN. Modelado en capas

FASE A: Modifica el proyecto Almacén para cumplir los siguientes requisitos:

1. Implementa los metodos setPrecio() de las clases Mueble, Televisor, Lavadora. Sigue las especificaciones del documento explicativo.

   Ejemplo de implementación en Televisor:

```
@Override
  public void setPrecio(double precioBase) {
     this.precio = precioBase;
      if (pulgadas > 40) {
          precio += precioBase * 0.1;
      }
      if ("Plasma".equals(tipo)) {
          precio -= precioBase * 0.1;
      }
  }
```

2. La gestión de las fechas está basado en jdk7, modifícalo para que utilice las clases de la jdk8 java.time.* . El formato de las **fechas** deberá ser como 04-agosto-17 (tanto al recogerlas como al mostrarlas).

   Se deberá usar DateTimeFormatter para definir el patrón y LocalDate.parse() para obtener la fecha y format() oara mostrarla.

3. Si al recoger una fecha salta una ParseException deberás recogerla y lanzar una excepción propia de la aplicación FormatoFechaErroneo mostrando el formato correcto y lo que se ha introducido mal. Pero el producto-lavadora sí debe crearse.

   En método validarFecha():

```
DateTimeFormatter patron = DateTimeFormatter.ofPattern("dd-MMMM-yy");
    LocalDate fechaformateada;

    try {
        fechaformateada = LocalDate.parse(fecha, patron);
    } catch (DateTimeParseException e) {
        throw new FormatoFechaErroneo(fecha, "dd-MMMM-yy");
    }
    return fechaformateada;
```

4. Solicita los datos de fabricante y marca en el caso de ser un electrodoméstico, y almacénalo donde corresponda.

   Pedir los datos antes de llamar pedirLavadora() y pedirTelevisar() y pasarles esos valores como parámetros, para dentro almacenarlos en los objetos creados.

5. Mejora la eficiencia temporal de los bucles de los métodos buscarProducto(), buscarCliente() e buscarVenta()

   Ejemplo en buscaProducto()

```
for (int i = 0; i < productos.size() && producto == null; i++) {
          if (productos.get(i).getId() == np) {
              producto = productos.get(i);
          }
}
```

6. Modifica buscarProducto() para que devuelva un objeto Producto y las llamadas a este método.

   Devolver el objeto producto que sea localizado. Y revisar todas las llamadas a ese método.

7. La solicitud de datos de cada tipo de cliente no es correcta. Averigua que pasa y soluciónalo.

   Las condiciones que evalúan la opción escrita por el usuario en pedirCliente() no son correctas.

8. No se visualiza una venta. Soluciónalo.

Falta un println para ver datos de la venta buscada.

9. El vendedor de una venta no se visualiza. Averigua porque y soluciónalo.

El vendedor no se está almacenando, se almacena el nombre del cliente en su lugar.

10. Para introducir una venta deben existir el producto y el cliente. Mejora el código utilizando los métodos buscaProducto() y buscaCliente().

Sustituir los bucles que buscan en clientes y productos por llamadas a buscaProducto(np) y buscaCliente(nc);

11. Cuando aparezca un enum debéis mostrar todos los posibles valores pero usando algún método del enum

Se debe utilizar el método .values() y mostrar su contenido

12. Al recoger datos de un cliente debéis calcular la letra del NIF y comprobar si es correcto. Si no lo fuera lanzar excepción y volver a solicitar el dato.

Método que calcula la letra del nif:

```
private String nifOK(String dni) throws Exception {
        boolean ok = true;
        String juegoCaracteres = "TRWAGMYFPDXBNJZSQVHLCKE";

        if (dni.length() == 8 || dni.length() == 9) {
            int modulo = Integer.parseInt(dni.substring(0, 8)) % 23;
            char letra = juegoCaracteres.charAt(modulo);
            dni = dni.substring(0, 8) + letra;
        } else {
            throw new Exception("Longitud de dni erronea");
        }
        return dni;
    }
```

Este método se debe invocar en pedirParticular() y controlar con un try-catch la Exception que se puede generar. Esto debe estar dentro de un bucle que se repetirá si se genera la exception.

FASE B:

Llegados a este punto podemos plantearnos si ¿**no hubiera sido más a apropiado hacer una clase service para Ventas, otra para Productos y otra para Clientes, cada una de las cuales gestionara su lista?** La respuesta a esa cuestión es que **SI**, es más apropiado. Modificaremos el proyecto creando tres clases *ProductosService, ClientesService y VentasService*, que sustituirán a *NegociosService*.

Deberás tener en cuenta, por ejemplo, que para eliminar un producto también hay que acceder a la lista de Ventas, que ahora estará en la clase VentasService. ¿Cómo lo implementarías?

posibilidad 1: implementar un método en la clase de servicio de ventas, que reciba el id de producto y borre las ventas asociadas a ese producto. Para poder acceder a este método cuando borramos un producto debemos tener acceso al objeto VentasService. Guárdate el objeto creado como atributo en la clase Producto.

¿Se te ocurre alguna más?

Recuerda que:

- para eliminar un cliente debemos eliminar las ventas de ese cliente primero

- para introducir una venta hay que acceder a productos y clientes

POSIBLE SOLUCIÓN

En la clase ProductoService:
- Se crea atributo de la clase VentasService.
- Le asignamos valor mediante el método set() en el constructor del MenuPrincipal, pasándole el objeto creado de VentasService
- En la clase VentasService creamos un método eliminarVentasxProducto(int numproducto)
- Se invoca desde el método de eliminarProducto(int numproducto), una vez eliminadas las ventas se puede eliminar el producto pedido.

En la clase Clienteservice sería parecido.

En la clase VentasService hariamos:
- Dos atributos para almacenar tanto Productos como Clientes
- Se asigna valor mediante los set() en el constructor de MenuPrincipal
- Al introducir la venta se utilizan los métodos buscaCliente() y buscaProducto() para localizar los objetos correspondientes.