Distribución de Productos en un Supermercado

Primera entrega: 18/11/2024

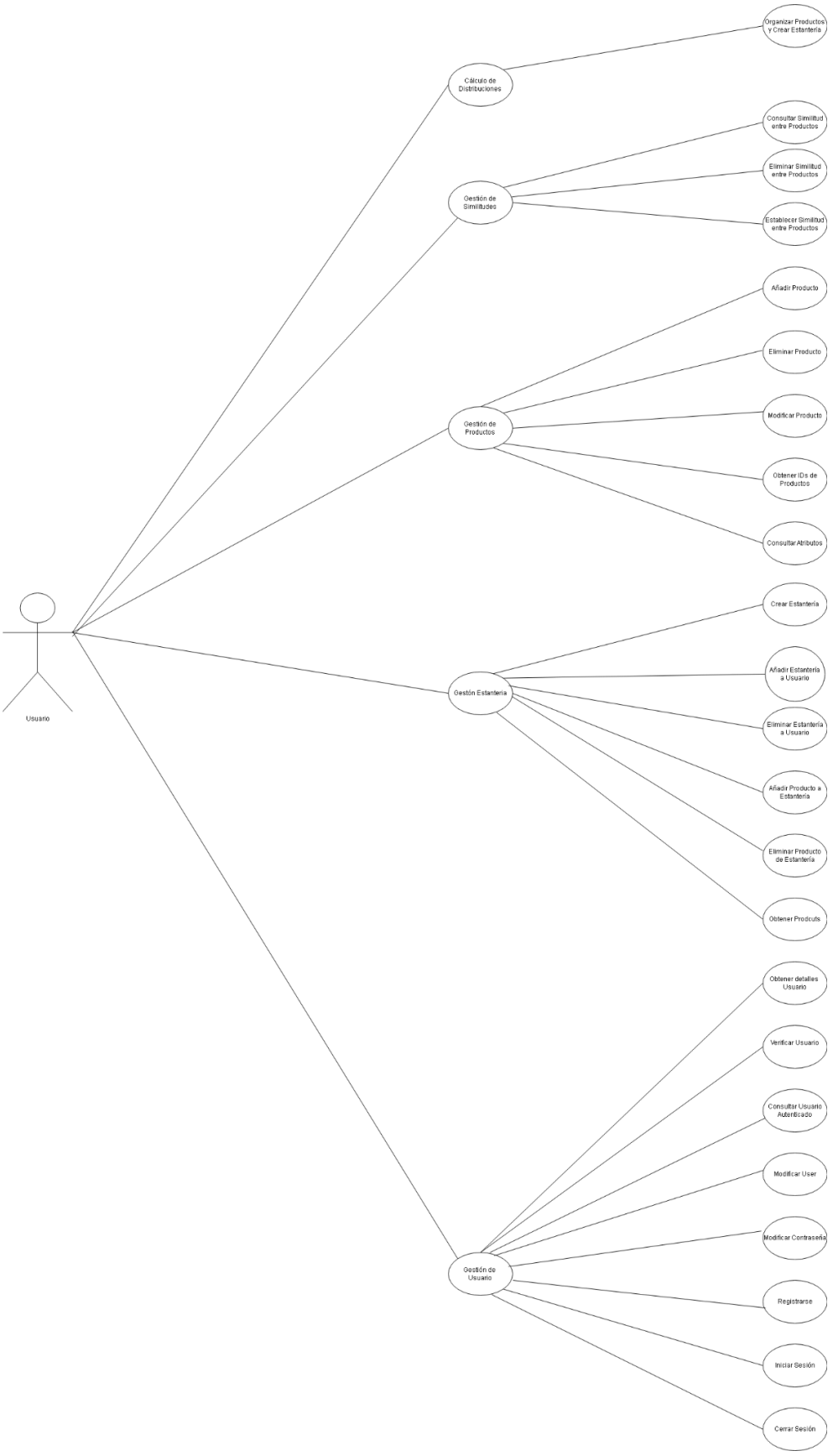
Versión: 1.1

Identificador del equipo: 31.3

Hug Capdevila: hug.capdevila@estudiantat.upc.edu
Levon Asatryan: levon.asatryan@estudiantat.upc.edu
Omar Cornejo: omar.antonio.cornejo@estudiantat.upc.edu
Nazarena Ponce: nazarena.ponce@estudiantat.upc.edu

Índice

1. Diagrama de casos de uso UML



1.1. Descripción casos de uso

1.1.1. Gestión de Usuarios

Nombre: Registrar Usuario
Función utilizada: `CtrlDomain.registerUser(String name, String username, String password)`
Actor: Usuario
Comportamiento (diálogo entre los actores y el sistema):

1. El usuario selecciona <Registrar Usuario> desde el estado inicial de la aplicación.
2. El sistema solicita al usuario un nombre, nombre de usuario y contraseña.
3. El usuario proporciona un nombre, un nombre de usuario y la contraseña.
4. El sistema valida los datos. Si los datos son correctos, el registro se completa y se crea un perfil asociado con el nombre de usuario.

Errores posibles:

1. El nombre de usuario ya está registrado, el sistema pide al usuario que ingrese uno nuevo (vuelve al estado inicial).

Nombre: Iniciar Sesión
Función utilizada: `CtrlDomain.loginUser(String username, String password)`
Actor: Usuario
Comportamiento (diálogo entre los actores y el sistema):

1. El usuario selecciona <Iniciar Sesión de Usuario> desde el estado inicial de la aplicación.
2. El sistema solicita al usuario un nombre de usuario y contraseña.
3. El usuario ingresa su nombre de usuario y su contraseña.
4. El sistema valida los datos. Si los datos son correctos, el sistema guarda los datos del usuario como UserSelected.

Errores posibles y cursos alternativos:

1. El usuario no está ingresado en el sistema (su nombre de usuario no existe), el sistema informa el error (vuelve al estado inicial).
2. La contraseña ingresada es incorrecta, el sistema pide al usuario que vuelva a ingresarla (vuelve al estado inicial).

Nombre: Cerrar Sesión
Función utilizada: `CtrlDomain.logoutUser()`
Actor: Usuario
Comportamiento (diálogo entre los actores y el sistema):

1. El usuario selecciona la opción <Cerrar Sesión de Usuario>.
2. El sistema cierra la sesión del usuario y vuelve al estado inicial de la aplicación.

Errores posibles y cursos alternativos:

1. No hay ningún usuario autenticado, el sistema informa del error y vuelve al estado inicial.

Nombre: Modificar Usuario
Función utilizada: `CtrlDomain.updateUsername(String oldUsername, String newUsername)`

Actor: Usuario
Comportamiento (diálogo entre los actores y el sistema):

1. El usuario selecciona la opción <Cambiar Nombre de Usuario>.
2. El sistema solicita al usuario su nombre de usuario actual y un nuevo nombre de usuario.
3. El usuario ingresa su nombre de usuario y su nuevo nombre de usuario.
4. El sistema valida los datos. Si los datos son correctos, se actualiza la información de UserSelected y del perfil con el usuario asociado, y vuelve al estado inicial.

Errores posibles y cursos alternativos:

1. El nombre de usuario nuevo ya existe, el sistema informa al usuario y no realiza ningún cambio (sale del caso de uso).

Nombre: Modificar Contraseña
Función utilizada: `CtrlDomain.updatePassword(String currentPassword, String newPassword)`
Actor: Usuario
Comportamiento (diálogo entre los actores y el sistema):

1. El usuario selecciona <Cambiar Contraseña>.
2. El sistema solicita al usuario su contraseña actual y la nueva contraseña.
3. El usuario ingresa su contraseña actual y la nueva contraseña.
4. El sistema comprueba la contraseña actual y, si es correcta, modifica los datos de UserSelected y los del perfil asociado al usuario.

Errores posibles y cursos alternativos:

1. La contraseña actual ingresada es incorrecta, el sistema informa al usuario y vuelve al estado inicial.

Nombre: Consultar Usuario Autenticado
Función utilizada: `CtrlDomain.getUsernameSelected()`
Actor: Usuario
Comportamiento (diálogo entre los actores y el sistema):

1. El usuario selecciona <Consultar Usuario Autenticado>.
2. El sistema obtiene el nombre de usuario de UserSelected, lo muestra en pantalla y vuelve al estado inicial.

Errores posibles y cursos alternativos:

1. No hay ningún usuario autenticado, el sistema informa del error y vuelve al estado inicial.

Nombre: Verificar Existencia de Usuario
Función utilizada: `CtrlDomain.existUser(String username)`
Actor: Usuario
Comportamiento (diálogo entre los actores y el sistema):

1. El usuario selecciona <Verificar Existencia de Usuario>.
2. El sistema pide al usuario que ingrese el nombre del usuario que desea verificar.
3. El usuario ingresa el nombre del usuario que desea verificar.
4. El sistema valida los datos y muestra en pantalla un mensaje diciendo si el usuario existe o no, luego vuelve al estado inicial.

Errores posibles y cursos alternativos:

Nombre: Obtener detalles usuario
Función utilizada: `getUserDetails()`
Actor: Usuario
Comportamiento (diálogo entre los actores y el sistema):

1. El usuario selecciona <Ver Detalles del Usuario Autenticado>.
2. El sistema obtiene el nombre y el nombre de usuario de UserSelected y los muestra por pantalla, luego vuelve al estado inicial.

Errores posibles y cursos alternativos:

1. No hay ningún usuario autenticado, el sistema informa del error y vuelve al estado inicial.

1.1.2. Gestión de Productos

Nombre: Añadir Producto
Función utilizada: `CtrlDomain.addProductUser(String name, int price)`
Actor: Usuario
Comportamiento (diálogo entre los actores y el sistema):

1. El usuario selecciona la opción <Añadir Producto al Usuario>.
2. El sistema solicita al usuario el nombre del producto y el precio del producto que desea añadir.
3. El usuario ingresa el nombre del producto y el precio.
4. El sistema valida los datos. Si los datos son correctos, el sistema actualiza el inventario de productos de UserSelected y del perfil asociado al usuario. También genera un ID para el producto y vuelve al estado inicial.

Errores posibles y cursos alternativos:

1. No hay ningún usuario autenticado, el sistema informa al usuario y no realiza ningún cambio (vuelve al estado inicial).
2. El producto ya existe en el inventario del usuario, el sistema informa al usuario y no realiza ningún cambio (vuelve al estado inicial).
3. El valor del precio ingresado es negativo, el sistema informa al usuario del error y no realiza ningún cambio (vuelve al estado inicial).

Nombre: Eliminar Producto
Función utilizada: `CtrlDomain.removeProductUser(String name)`
Actor: Usuario
Comportamiento (diálogo entre los actores y el sistema):

1. El usuario selecciona la opción <Eliminar Producto del Usuario>.
2. El sistema solicita al usuario el nombre del producto que se desea eliminar.
3. El usuario ingresa el nombre del producto.
4. El sistema valida los datos (se asegura de que el producto exista en el inventario del usuario).
5. Si el producto existe en el inventario, el sistema lo elimina y actualiza el inventario de UserSelected y del perfil asociado al usuario, luego vuelve al estado inicial.

Errores posibles y cursos alternativos:

1. No hay ningún usuario autenticado, el sistema informa al usuario y no realiza ningún cambio (vuelve al estado inicial).

2. El producto no existe en el inventario del usuario, el sistema informa al usuario y no realiza ningún cambio (vuelve al estado inicial).

Nombre: Modificar Producto
Función utilizada: `CtrlDomain.updateProductUser(String name, int newPrice)`
Actor: Usuario
Comportamiento (diálogo entre los actores y el sistema):
1. El usuario selecciona la opción <Actualizar Precio de Producto del Usuario>.
2. El sistema solicita al usuario el nombre del producto y el nuevo valor del precio.
3. El usuario ingresa el nombre del producto y el precio modificado.
4. El sistema valida los datos. Si son correctos, actualiza el inventario de productos de UserSelected y del perfil asociado al usuario y vuelve al estado inicial.

Errores posibles y cursos alternativos:
1. No hay ningún usuario autenticado, el sistema informa al usuario y no realiza ningún cambio (vuelve al estado inicial).
2. El precio es un valor negativo, el sistema informa al usuario del error y no realiza ningún cambio (vuelve al estado inicial).

Nombre: Consultar Atributos
Función utilizada: `CtrlDomain.getProductAttributesUser(String name)`
Actor: Usuario
Comportamiento (diálogo entre los actores y el sistema):
1. El usuario selecciona la opción <Obtener Atributos de Producto>.
2. El sistema solicita al usuario que ingrese el nombre del producto que desea consultar.
3. El usuario ingresa el nombre del producto del cual desea saber los atributos.
4. El sistema busca el producto en el inventario de UserSelected y, si lo encuentra, muestra por pantalla el ID, el nombre y el precio del producto.

Errores posibles y cursos alternativos:
1. No hay ningún usuario autenticado, el sistema informa al usuario y no realiza ningún cambio (vuelve al estado inicial).

Nombre: Obtener IDs de Productos
Función utilizada: `getProductIdsForUser(String username)`
Actor: Usuario
Comportamiento (diálogo entre los actores y el sistema):
1. El usuario selecciona la opción <Obtener Ids de Productos del Usuario>.
2. El sistema obtiene los datos del inventario de Productos de UserSelected y muestra por pantalla los IDs de los productos almacenados.

Errores posibles y cursos alternativos:
1. No hay ningún usuario autenticado, el sistema informa al usuario y no realiza ningún cambio (vuelve al estado inicial).

1.1.3. Gestión de Similitudes

Nombre: Establecer Similitud entre Productos
Función utilizada: `CtrlDomain.setProductSimilarityForUser(String productName1, String productName2, double similarity)`
Actor: Usuario
Comportamiento (diálogo entre los actores y el sistema):

1. El usuario selecciona la opción <Establecer Similitud entre Productos>.
2. El sistema solicita al usuario el nombre de dos productos y el valor de la similitud.
3. El usuario ingresa los datos solicitados.
4. El sistema valida los datos. Si los datos son correctos, actualiza la similitud relacionada con ambos productos y vuelve al estado inicial.

Errores posibles y cursos alternativos:

1. No hay ningún usuario autenticado, el sistema informa al usuario y no realiza ningún cambio (vuelve al estado inicial).
2. El valor de la similitud está fuera del rango [0,1], el sistema informa al usuario del error y no realiza ningún cambio (vuelve al estado inicial).
3. Al menos uno de los productos no existe en el inventario de UserSelected, el sistema informa al usuario indicando el nombre del producto que no existe y no realiza ningún cambio (vuelve al estado inicial).

Nombre: Eliminar Similitud entre Productos
Función utilizada: `CtrlDomain.removeProductSimilarityForUser(String productName1, String productName2)`
Actor: Usuario
Comportamiento (diálogo entre los actores y el sistema):

1. El usuario selecciona la opción <Eliminar Similitud entre Productos>.
2. El sistema solicita al usuario el nombre de los productos cuya similitud desea eliminar.
3. El usuario ingresa los datos solicitados.
4. El sistema valida los datos. Si son correctos, actualiza los valores de la similitud poniendo un -1 en la relación entre los dos productos seleccionados y vuelve al estado inicial.

Errores posibles y cursos alternativos:

1. No hay ningún usuario autenticado, el sistema informa al usuario y no realiza ningún cambio (vuelve al estado inicial).
2. Al menos uno de los productos no existe en el inventario de UserSelected, el sistema informa al usuario y no realiza ningún cambio (vuelve al estado inicial).
3. No existe relación entre los dos productos ingresados, el sistema informa del error al usuario y no realiza ningún cambio (vuelve al estado inicial).

Nombre: Consultar Similitud entre Productos
Función utilizada: `CtrlDomain.getProductSimilarityForUser(String productName1, String productName2)`
Actor: Usuario
Comportamiento (diálogo entre los actores y el sistema):

1. El usuario selecciona <Obtener Similitud entre Productos>.
2. El sistema solicita al usuario el nombre de los productos cuya similitud desea consultar.

3. El usuario ingresa los datos solicitados.
4. El sistema valida los datos. Si los datos son correctos, muestra en pantalla la similitud entre los dos productos y vuelve al estado inicial.

Errores posibles y cursos alternativos:

1. No hay ningún usuario autenticado, el sistema informa al usuario y no realiza ningún cambio (vuelve al estado inicial).
2. Al menos uno de los productos no existe en el inventario de UserSelected, el sistema informa al usuario indicando el nombre del producto que no existe y no realiza ningún cambio (vuelve al estado inicial).
3. No existe relación entre los dos productos ingresados, el sistema informa del error al usuario y no realiza ningún cambio (vuelve al estado inicial).

1.1.4. Gestión de Estanterías

Nombre: Añadir Estantería a Usuario
Función utilizada: `CtrlDomain.addShelfForUser(String shelfName)`
Actor: Usuario
Comportamiento (diálogo entre los actores y el sistema):

1. El usuario selecciona la opción <Crear Estantería>.
2. El sistema solicita al usuario que ingrese el nombre de la estantería que desea crear.
3. El usuario ingresa el nombre de la nueva estantería.
4. El sistema valida los datos. Si el nombre de la estantería no existía previamente, actualiza las estanterías de UserSelected y del perfil asociado al usuario autenticado, y vuelve al estado inicial.

Errores posibles y cursos alternativos:

1. No hay ningún usuario autenticado, el sistema informa al usuario y no realiza ningún cambio (vuelve al estado inicial).
2. Existe una estantería con el mismo nombre, el sistema informa al usuario del error y no realiza ningún cambio (vuelve al estado inicial).

Nombre: Eliminar Estantería de Usuario
Función utilizada: `CtrlDomain.removeShelfForUser(String shelfName)`
Actor: Usuario
Comportamiento (diálogo entre los actores y el sistema):

1. El usuario selecciona la opción <Eliminar Estantería>.
2. El sistema solicita al usuario que ingrese el nombre de la estantería que desea eliminar.
3. El usuario ingresa el nombre de la estantería que quiere eliminar.
4. El sistema valida los datos. Si los datos son correctos, actualiza las estanterías de UserSelected y las del perfil asociado al usuario autenticado, y vuelve al estado inicial.

Errores posibles y cursos alternativos:

1. No hay ningún usuario autenticado, el sistema informa al usuario y no realiza ningún cambio (vuelve al estado inicial).
2. No existe la estantería, el sistema informa al usuario del error y no realiza ningún cambio (vuelve al estado inicial).

Nombre: Añadir Producto a Estantería
Función utilizada: `CtrlDomain.addProductToShelfForUser(String shelfName, int idProduct)`
Actor: Usuario
Comportamiento (diálogo entre los actores y el sistema):

1. El usuario selecciona la opción <Añadir Producto a Estantería>.
2. El sistema solicita al usuario que introduzca el nombre de la estantería y el producto que desea añadir a ella.
3. El usuario ingresa el nombre de la estantería y del producto.
4. El sistema valida los datos comprobando que la estantería y el producto se encuentren en el inventario de UserSelected. Si los datos son correctos, asocia el producto con la estantería, lo coloca en la primera posición permitida y vuelve al estado inicial.

Errores posibles y cursos alternativos:

1. No hay ningún usuario autenticado, el sistema informa al usuario y no realiza ningún cambio (vuelve al estado inicial).
2. No existe la estantería, el sistema informa al usuario del error y no realiza ningún cambio (vuelve al estado inicial).
3. No existe el producto en el inventario del usuario, el sistema informa al usuario y no realiza ningún cambio (vuelve al estado inicial).
4. El producto ya existe en la estantería, el sistema informa al usuario y no realiza ningún cambio (vuelve al estado inicial).

Nombre: Obtener Productos
Función utilizada: `CtrlDomain.getProductsInUserShelf(String shelfName)`
Actor: Usuario autenticado
Comportamiento (diálogo entre los actores y el sistema):

1. El usuario selecciona la opción <Obtener Productos de una Estantería>.
2. El sistema solicita al usuario que introduzca el nombre de la estantería que desea consultar.
3. El usuario ingresa el nombre de la estantería.
4. El sistema llama a la función `getProductsInUserShelf(shelfName)` para obtener la lista de productos en la estantería seleccionada.
5. Si la estantería existe en el inventario de UserSelected y el usuario está autenticado, el sistema muestra la lista de productos en pantalla y sale del caso de uso.

Errores posibles y cursos alternativos:

1. No hay ningún usuario autenticado, el sistema informa al usuario y no realiza ningún cambio (vuelve al estado inicial).
2. Si la estantería no existe para UserSelected, el sistema muestra un mensaje de error indicando que no se ha encontrado la estantería y vuelve al estado inicial.
3. Si el producto no existe para UserSelected, el sistema muestra un mensaje de error indicando que no se ha encontrado el producto y vuelve al estado inicial.

Nombre: Eliminar Producto de Estantería
Función utilizada: `CtrlDomain.removeProductFromShelfForUser(String shelfName, int idProduct)`
Actor: Usuario
Comportamiento (diálogo entre los actores y el sistema):

1. El usuario selecciona la opción <Eliminar Producto de Estantería>.
2. El sistema solicita al usuario que introduzca el nombre de la estantería a modificar y el producto que quiere eliminar.
3. El usuario ingresa el nombre de la estantería y del producto.
4. El sistema valida los datos. Si los datos son correctos, elimina el producto de la estantería.

Errores posibles y cursos alternativos:

1. No hay ningún usuario autenticado, el sistema informa al usuario y no realiza ningún cambio (vuelve al estado inicial).
2. El producto no existe en la estantería, el sistema informa al usuario y no realiza ningún cambio (vuelve al estado inicial).
3. La estantería no existe en el inventario de UserSelected, el sistema informa al usuario y no realiza ningún cambio (vuelve al estado inicial).

1.1.5. Cálculo de Distribuciones

Nombre: Organizar Productos y Crear Estantería
Función utilizada: `organizeProductsAndCreateShelf(String shelfName, AlgorithmType algorithmType)`
Actor: Usuario
Comportamiento (diálogo entre los actores y el sistema):

1. El usuario selecciona la opción <Organizar Productos y Crear Estantería>.
2. El sistema solicita al usuario que introduzca el nombre de la nueva estantería.
3. El usuario introduce el nombre de la nueva estantería.
4. El sistema valida el nombre de la estantería asegurándose de que no exista otra con el mismo nombre. Si el dato es correcto, solicita al usuario que elija el algoritmo a utilizar.
    a. (1) Two Approximation.
    b. (2) Brute Force.
5. El usuario selecciona el algoritmo deseado.
6. Según la elección del algoritmo:
    a. *Two Approximation*
        i. El sistema verifica que se cumpla la propiedad de desigual triangular.
        ii. Si se cumple, organiza los productos de la lista de productos de UserSelected y crea la estantería.
    b. *Brute Force:*
        i. El sistema verifica el tamaño de la lista de productos de UserSelected.
        ii. Si tiene 8 elementos o más muestra un mensaje informando al usuario que la estantería puede tardar tiempo en organizarse. Crea la estantería con la distribución organizada por el algoritmo.
        iii. Organiza los productos y crea la estantería.

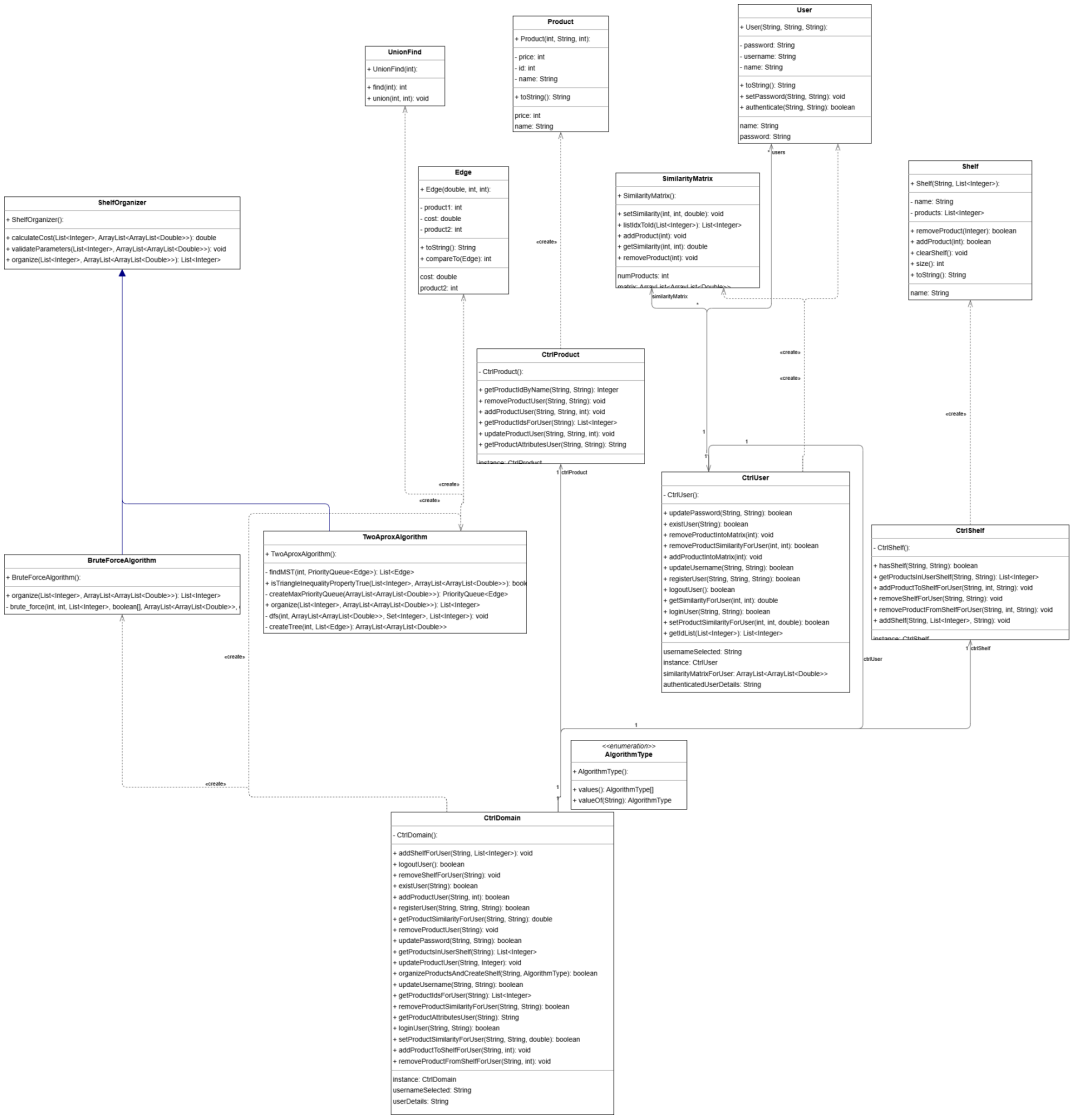
Errores posibles y cursos alternativos:

1. No hay ningún usuario autenticado, el sistema informa al usuario y no realiza ningún cambio (vuelve al estado inicial).
2. El nombre de la estantería ya existe, el sistema informa al usuario y no realiza ningún cambio (vuelve al estado inicial).

3. Se ha escogido el algoritmo Two Approximation pero la propiedad de desigualdad triangular no se cumple para las similitudes, el sistema muestra un mensaje de error informando al usuario y vuelve al estado inicial sin realizar ningún cambio.
4. El usuario introduce una opción inválida para elegir el algoritmo, el sistema informa del error y no realiza ningún cambio (vuelve al estado inicial).

2. Diagrama del modelo conceptual

2.1. Diseño del diagrama del modelo conceptual



Restricciones textuales:

1. Una estantería no puede tener productos duplicados.
2. El precio de un producto debe ser mayor que cero.
3. Los nombres de los productos deben ser únicos dentro del sistema.
4. Una conexión (`Edge`) debe conectar dos productos diferentes.
5. El costo de una conexión debe ser positivo.
6. Cada usuario debe tener un nombre de usuario único.
7. La contraseña no puede ser null.
8. Un usuario no puede tener dos estanterías con el mismo nombre.
9. Antes de organizar una estantería, debe haber al menos un producto en la lista.
10. Los valores en la diagonal principal de la matriz de similitud deben ser iguales a 1.

2.2. Descripción de las clases

2.2.1. Clase User

Descripción: La clase `User` representa a un usuario de la aplicación que puede gestionar un inventario de productos y una colección de estanterías. Además, puede establecer similitudes entre sus productos.

Cardinalidad:

- Un usuario puede poseer varios productos.
- Un usuario puede poseer múltiples estanterías.
- Un producto puede estar en varias estanterías de un mismo usuario.

Atributos:

- **name:** (`String`) Nombre completo del usuario. Este es un atributo final.
- **username:** (`String`) Nombre único de usuario.
- **password:** (`String`) Contraseña del usuario.

Relaciones:

- Relación con la clase `CtrlUser`.

2.2.2. Clase Product

Descripción: La clase `Product` representa un producto en la aplicación. Cada producto tiene una identificación única, un nombre y un precio. Esta clase permite gestionar productos y realizar modificaciones de atributos específicos. Los productos pueden ser añadidos a inventarios de usuarios y a estanterías.

Cardinalidad:

- Un usuario puede poseer múltiples productos.
- Una estantería puede contener múltiples productos.

Atributos:

- **id:** (`int`) Identificación única del producto. Este atributo es final y no puede cambiarse después de la creación del producto.
- **name:** (`String`) Nombre del producto.
- **price:** (`int`) Precio actual del producto.

Relaciones:

- Relación con la clase `CtrlProduct`.

2.2.3. Clase SimilarityMatrix

Descripción: La clase `SimilarityMatrix` modela una matriz de similitudes entre productos, donde cada producto tiene un identificador único y su similitud con otros productos se almacena en una estructura de datos.

Cardinalidad:

- Un usuario puede tener varias estanterías, por lo que pueden haber múltiples instancias de la clase.

Atributos:

- **idToIndex**: Mapa que asocia IDs de productos a índices dentro de la matriz.
- **indexToId**: Mapa que asocia índices de productos a IDs dentro de la matriz.
- **idToSimilarities**: Mapa que guarda las similitudes de cada producto con otros productos.

Relaciones:

- Relación con la clase `CtrlUser`.

2.2.4. Clase Shelf

Descripción: La clase `Shelf` representa una estantería que contiene una lista de productos, permitiendo organizar y gestionar estos productos para un usuario.

Cardinalidad:

- Un usuario puede tener varias estanterías.
- Una estantería puede contener varios productos.
- Un producto puede estar presente en múltiples estanterías.

Atributos:

- **name:** (`String`) Nombre único que identifica la estantería.
- **products:** (`List<Integer>`) Lista de IDs de los productos contenidos en la estantería.

Relaciones:

- Relación con la clase `CtrlShelf`.

2.2.5. Clase CtrlDomain

Descripción: La clase `CtrlDomain` actúa como el controlador principal de la aplicación, integrando múltiples componentes y funcionalidades, tales como la gestión de usuarios, productos, estanterías y algoritmos de organización. Sigue el patrón de diseño Singleton para garantizar una única instancia en el sistema y facilitar la gestión centralizada de los controladores secundarios (`CtrlUser`, `CtrlProduct`, `CtrlShelf`). Proporciona una interfaz unificada para las operaciones más comunes del dominio.

Cardinalidad:

- CtrlDomain contiene una única instancia de CtrlUser.
- CtrlDomain contiene una única instancia de CtrlProduct.
- CtrlDomain contiene una única instancia de CtrlShelf.
- CtrlDomain utiliza un único algoritmo activo para ejecutar operaciones específicas.
- Solo puede existir una instancia de CtrlDomain debido a su implementación como Singleton.

Atributos:

- **ctrlProduct:** (`CtrlProduct`) Controlador para la gestión de productos.
- **ctrlShelf:** (`CtrlShelf`) Controlador para la gestión de estanterías.
- **ctrlUser:** (`CtrlUser`) Controlador para la gestión de usuarios y autenticación.
- **singletonObject:** (`CtrlDomain`) Instancia única de CtrlDomain que sigue el patrón Singleton.

Relaciones:
- Relación con la clase `CtrlUser`.
- Relación con la clase `CtrlProduct`.
- Relación con la clase `CtrlShelf`.

2.2.6. Clase CtrlUser

Descripción: La clase `CtrlUser` es responsable de gestionar las operaciones relacionadas con los usuarios, incluida su autenticación y registro, así como la administración de las matrices de similitud entre productos. Esta clase sigue el patrón de diseño Singleton para asegurar que solo exista una instancia a lo largo de la ejecución de la aplicación. Además, ofrece métodos para gestionar tanto la autenticación de usuarios como las similitudes entre productos del usuario autenticado.

Cardinalidad:

- La clase `CtrlUser` tiene una única instancia a lo largo del ciclo de vida de la aplicación (patrón Singleton).

Atributos:

- **users**: (`Map<String, User>`) Mapa que asocia el nombre de usuario con un objeto `User`.
- **similarityMatrix**: (`Map<String, SimilarityMatrix>`) Mapa que asocia el nombre de usuario con la matriz de similitud correspondiente.
- **usrSelected**: (`String`) Cadena que almacena el nombre de usuario del usuario autenticado.
- **singletonObject**: (`CtrlUser`) Instancia única de la clase `CtrlUser` (patrón Singleton).

Relaciones:
- Relación con la clase `User`.
- Relación con la clase `CtrlDomain`.

2.2.7. Clase CtrlProduct

Descripción: La clase `CtrlProduct` gestiona el inventario de productos de los usuarios. Cada usuario tiene su propio conjunto de productos, que están almacenados en un mapa. Los productos pueden ser añadidos, eliminados, modificados o consultados a través de los métodos de esta clase.

Cardinalidad:

- La clase `CtrlProduct` es un singleton, lo que significa que solo existe una instancia de esta clase en el sistema.

Atributos:

- **products**: (`Map<String, HashMap<String, Product>>`) Almacena los productos de cada usuario. La clave es el nombre del usuario, y el valor es un mapa de productos asociados con ese usuario, siendo la clave el nombre del producto.
- **singletonObject**: (`CtrlProduct`) Instancia única de la clase, utilizada para implementar el patrón Singleton.
- **productIdCounter**: (`int`) Contador de ID de productos que se asigna a cada nuevo producto. Empieza en 1.

Relaciones:

- Relación con la clase `Product`.
- Relación con la clase `CtrlDomain`.

2.2.8. Clase CtrlShelf

Descripción: La clase `CtrlShelf` gestiona las estanterías de los usuarios, permitiendo añadir, eliminar y consultar estanterías, así como manipular los productos que contienen. Es una clase singleton, lo que garantiza que solo haya una instancia activa en el sistema.

Cardinalidad:

- Es una clase singleton, por lo que solo existe una instancia.

Atributos:

- **shelves:** (`Map<String, HashMap<String, Shelf>>`) Mapa que almacena las estanterías de cada usuario. La clave principal es el nombre del usuario, y el valor es otro mapa donde la clave es el nombre de la estantería y el valor es la estantería en sí.
- **singletonObject:** (`CtrlShelf`) Instancia única de la clase, utilizada para implementar el patrón Singleton.

Relaciones:

- Relación con la clase `Shelf`.
- Relación con la clase `CtrlDomain`.

2.2.9. Clase Edge

Descripción: La clase `Edge` representa una arista en un grafo ponderado, utilizada para modelar la relación entre dos productos con un costo asociado (su similitud)

Cardinalidad:

- La clase `Edge` se usa múltiples veces en el cálculo del MST durante el algoritmo, por lo que tendremos varias instancias.

Atributos:

- **cost**: (`double`) El costo de la arista, que representa la similitud entre dos productos.

- **product1**: (`int`) Identificador del primer producto conectado por esta arista.
- **product2**: (`int`) Identificador del segundo producto conectado por esta arista.

Relaciones:
- Relación con la clase `TwoAproxAlgorithm`

2.2.10. Clase UnionFind

Descripción: La clase `UnionFind` implementa la estructura de datos Union-Find que permite gestionar conjuntos de elementos particionados en subconjuntos disjuntos. Esto es especialmente útil para implementar de forma eficiente el algoritmo de Kruskal.

Cardinalidad:

- Creamos una nueva instancia de la clase `UnionFind` cada vez que ejecutamos el algoritmo de 2Aproximación.

Atributos:

- **nodes**: (`int[]`) Almacena en qué conjunto se encuentran los nodos de la estructura de datos. Si hay un valor negativo en el índice i, siendo i un índice válido, significa que el nodo i es el padre de su conjunto. Además el valor absoluto del contenido de la celda será el número de elementos que tiene ese conjunto.

Relaciones:
- Relación con la clase `TwoAproxAlgorithm`.

2.2.11. Clase ShelfOrganizer

Descripción: La clase abstracta `ShelfOrganizer` define la estructura básica de las clases que tienen el objetivo de organizar productos en un orden específico.

Cardinalidad:

- La clase `ShelfOrganizer` es abstracta, por lo que no habrá instancias de esta. Lo que sí que tendremos serán instancias de las clases que heredan de esta.

Relaciones:
- Relación con la clase `CtrlDomain`.

2.2.12. Clase TwoAproxAlgorithm

Descripción: La clase `TwoAproxAlgorithm` hereda de `ShelfOrganizer` y implementa un algoritmo para encontrar una aproximación de la distribución óptima, es decir, maximizar las similitudes entre los productos adyacentes.

Cardinalidad:

- La clase `TwoAproxAlgorithm` tendrá una instancia en CtrlDomain.

Relaciones:

- Relación indirecta con la clase `CtrlDomain`.

2.2.13. Clase BruteForceAlgorithm

Descripción: La clase `BruteForceAlgorithm` hereda de `ShelfOrganizer` y implementa un algoritmo de fuerza bruta para encontrar la distribución óptima de productos, es decir, maximizar las similitudes entre los productos adyacentes.

Cardinalidad:

- La clase `TwoAproxAlgorithm` tendrá una instancia en CtrlDomain.

Relaciones:

- Relación indirecta con la clase `CtrlDomain`.

3. Relaciones de las clases implementadas por miembro del grupo

Clase	Controlador

El mismo miembro del grupo que ha hecho la clase también ha hecho los tests y documentación de dicha clase

HUG	OMAR	LEVON	NAZARENA
Edge	User	Product	Shelf
SimilarityMatrix	SimilarityMatrix	CtrlProduct	CtrlShelf
UnionFind	CtrlUser	Clase CtrlDomain: Funciones relacionadas con Gestión de Productos	Clase CtrlDomain: Funciones relacionadas con Gestión de Estanterías
BruteForceAlgorithm	CtrlDomainDriver	ProductDriver	ShelfDriver + CtrlDomainDriver funciones relacionadas con gestión de Estanterías
TwoAproxAlgorithm	SimilarityMatrixDriver	Juegos de Prueba	Descripción de los casos de uso
ShelfOrganizer	Excepciones	CtrlDomainDriver: Corrección de errores y añadir funciones restantes	Descripción de las clases (User, Product, SimilarityMatrix, Shelf, CtrlDomain, CtrlUser, CtrlProduct, CtrlShelf)
Enum AlgorithmType	Juegos de Prueba		Juegos de Prueba
TriangleInequalityViolationException	Casos de Uso		
Clase CtrlDomain: Funciones que se relacionan con Algoritmo	Diagrama de entrega		
UML: Parte relacionada con algoritmo	Javadoc		
Ficheros + Documento con las instrucciones para ejecutar todos los tests unitarios			
Estructures de Dades i Algoritmes utilitzats (Edge, UnionFind, BruteForceAlgorithm,TwoAproxAlgorithm, SimMatrix, ShelfOrganizer)			

4. Estructura de datos y algoritmos utilizados

A continuación explicaremos cada una de las estructuras de datos principales junto con los algoritmos más relevantes que hemos considerado para nuestra aplicación.

4.1. Clase SimilarityMatrix

En un principio pensamos en hacer una implementación trivial de la matriz de similitudes, es decir, que su atributo sea precisamente una matriz de doubles. No obstante, a medida que avanzamos en el proyecto pudimos ver que esta solución, todo y que modelaba bien lo que queríamos representar, se integraba mal con las otras partes de nuestra aplicación, por tanto, tuvimos que idear una nuevo modo de mantener una representación válida a la vez que era coherente con todos los demás componentes.

Estructuras de datos:
Para implementar la matriz de similitudes, elegimos una estructura basada en **tres HashMaps** que permiten una representación eficiente y flexible de la matriz de similitud entre productos. Esto nos permitió añadir la funcionalidad que permite al usuario añadir y quitar productos de forma dinámica una vez la estantería ha sido creada.

HashMaps en Java:
Es una estructura de datos que almacena pares clave-valor, permitiendo acceso rápido O(1) a valores a través de sus claves. Utiliza una función de hash para mapear claves a índices en una tabla hash, resolviendo colisiones mediante listas enlazadas o árboles binarios . No garantiza el orden de los elementos y permite una clave y múltiples valores nulos

Coste de las operaciones:

Inserción (put): Tiempo promedio O(1); en el peor de los casos, O(n) si todas las claves colisionan y crean una lista enlazada larga o árbol.

Búsqueda (get): Tiempo promedio O(1); en el peor de los casos, O(n) por colisiones excesivas o conversión de listas largas a árboles.

Eliminación (remove): Tiempo promedio O(1); en el peor de los casos, O(n) debido a colisiones.

Redimensionamiento (resize): Se dispara automáticamente cuando superamos el 75% de su capacidad. El coste es O(n), ya que redistribuye todas las claves en una tabla nueva más grande.

4.2. Clase UnionFind

La clase implementa la estructura de datos Union-Find o Disjoint Set Union (DSU), diseñada para gestionar particiones de elementos en subconjuntos disjuntos. Es útil para manejar eficientemente operaciones de unión y búsqueda en problemas de conectividad y partición de conjuntos.

Estructura de Datos:

La estructura se implementa usando un arreglo nodes, donde:

- Cada índice representa un elemento o nodo en el conjunto.
- Valores negativos en el arreglo indican el tamaño del conjunto (raíz) del elemento. Si el valor en nodes[i] es negativo, el nodo i es una raíz y su valor representa el tamaño del conjunto que encabeza.
- Valores no negativos almacenan la referencia al padre directo del nodo, manteniendo la estructura del árbol de conjuntos.

Algoritmos:

Compresión de Caminos (Path Compression):

- La función `find(int n)` busca la raíz representativa del conjunto al que pertenece un nodo n.
- Al encontrar la raíz, asigna esta raíz como padre de cada nodo en el camino de búsqueda, aplanando la estructura del árbol. Esto reduce la profundidad de los árboles, mejorando la eficiencia en futuras operaciones de búsqueda.

Unión por Tamaño (Union by Size):

- La función `union(int a, int b)` combina los conjuntos de dos nodos a y b.
- Se selecciona como raíz del conjunto resultante la raíz del conjunto más grande, uniendo el conjunto más pequeño al más grande. Esto minimiza la altura del árbol resultante, optimizando la estructura para futuras búsquedas.

Estas optimizaciones permiten realizar tanto las operaciones `find` como `union` en casi tiempo constante, amortizado **O(α(n))**, donde α es la inversa de la función de Ackermann, que crece extremadamente lento, haciendo que el costo sea casi constante en la práctica.

4.3. Clase BruteForceAlgorithm

La clase implementa un enfoque de fuerza bruta para organizar productos en un ciclo Hamiltoniano que maximiza el costo total de similitud entre productos adyacentes.

Algoritmos:

Fuerza Bruta para Ciclo Hamiltoniano Máximo:

- Permutaciones Complejas: Este algoritmo explora todas las posibles permutaciones de los productos, calculando el costo total de similitud de cada ciclo generado.
- Condición de Ciclo: Para cada permutación, se considera un ciclo cerrado (regresando al primer producto) y se calcula su costo total.
- Comparación y Almacenamiento de Máximo: Si el costo de la permutación actual supera el costo máximo registrado, se almacena como el nuevo ciclo de mayor similitud.

Complejidad Factorial:

- Debido a su complejidad O(n!), este enfoque solo es adecuado para conjuntos pequeños de productos, ya que el tiempo de ejecución aumenta factorialmente con el número de productos.
- Este enfoque garantiza encontrar la solución óptima al problema, pero su alta complejidad limita su uso a problemas de menor escala.

4.4. Clase TwoAproxAlgorithm

La clase implementa un algoritmo de aproximación 2 para organizar productos en un ciclo Hamiltoniano de alto costo, optimizando la similitud entre productos adyacentes.

Algoritmos:

Propiedad de Desigualdad Triangular

Antes de comenzar, se verifica que la matriz inversa de similitud cumple con la propiedad de desigualdad triangular (`isTriangleInequalityPropertyTrue`). Si no se cumple, el algoritmo puede no producir una buena aproximación.

Kruskal Modificado para MST

- Construcción del MST: Utiliza el algoritmo de Kruskal para construir un MST sobre el grafo de similitudes. La estructura UnionFind permite unir conjuntos y encontrar componentes conectados eficientemente.
- Cola de Prioridad Máxima: Almacena las aristas inversas ordenadas por similitud ascendente, permitiendo seleccionar las más fuertes en cada paso. Los conjuntos se unen por similitud hasta que se completa el MST.

DFS para Ciclo Hamiltoniano Aproximado

- Tras construir el MST, realiza una búsqueda en profundidad (DFS) para generar un ciclo Hamiltoniano.
- Ciclo Cerrado: Finalmente, el ciclo se cierra conectando el último producto con el inicial, formando el ciclo Hamiltoniano deseado.
- Este enfoque basado en MST y DFS permite obtener un ciclo Hamiltoniano de alta calidad en cuanto a similitud, con una complejidad de aproximación correcta.

4.5. Clase User

La clase `User` fue diseñada para representar un usuario en el sistema, incluyendo información básica como el nombre completo, un nombre de usuario único y una contraseña. Aunque su estructura puede parecer sencilla, la implementación incluye validaciones y métodos clave para garantizar seguridad y consistencia en el manejo de datos de usuario.

Estructuras de Datos:

Atributos:

- `name`: Un `String` que representa el nombre completo del usuario. Este atributo es inmutable y se define al crear el objeto.
- `username`: Un `String` que almacena el nombre único de usuario. Puede actualizarse, pero debe cumplir con restricciones de no nulidad ni vacío.
- `password`: Un `String` que almacena la contraseña del usuario. Su modificación está sujeta a validaciones estrictas.

Algoritmos y Costes:

Constructor de User
El constructor valida que todos los parámetros sean no nulos y no vacíos antes de inicializar los atributos.

- Coste: **O(1)**, ya que las validaciones y asignaciones son operaciones constantes.

Método de Autenticación (`authenticate`)
Este método compara las credenciales proporcionadas con las almacenadas.

- Coste: **O(n)**, donde *n* es la longitud de las cadenas `username` y `password`. La comparación de cadenas implica verificar cada carácter.

Actualización de Nombre de Usuario (`setUsername`)
Actualiza el nombre de usuario tras validar que no sea nulo ni vacío.

- Coste: **O(1)**.

Actualización de Contraseña (`setPassword`)
Este método realiza varias validaciones:

1. Comprueba que las contraseñas proporcionadas no sean nulas ni vacías.
2. Verifica que la contraseña actual coincida con la almacenada.
3. Actualiza la contraseña si todas las condiciones son satisfechas.
- Coste: **O(n)**, ya que implica comparar cadenas y asignar la nueva contraseña.

Representación como Cadena (`toString`)
Genera una representación textual del usuario.

- Coste: **O(m)**, donde *m* es la longitud de la salida generada.

4.6. Clase Product

En un principio, la clase `Product` fue diseñada como una estructura sencilla con atributos para almacenar información básica de un producto (ID, nombre y precio). Este enfoque permite gestionar los productos de manera directa y eficiente dentro del dominio de la aplicación.

No obstante, al avanzar en el desarrollo, identificamos ciertos requerimientos adicionales, como la necesidad de garantizar que los datos fueran siempre válidos al crearse o actualizarse. Por ello, añadimos validaciones estrictas en el constructor y en los métodos de modificación de atributos.

Estructuras de Datos:

La clase `Product` utiliza tipos de datos primitivos (`int` y `String`) para representar los atributos del producto. Aunque estas estructuras son simples, su correcta integración y validación son esenciales para garantizar la robustez de los datos.

Atributos:

- `id`: Representado como un `int`, este campo asegura unicidad y es inmutable una vez asignado.
- `name`: Un `String` que almacena el nombre del producto, asegurando siempre un valor no nulo ni vacío mediante validación.
- `price`: Representado como un `int` para evitar problemas con la precisión y validado para garantizar que sea un valor no negativo.

Algoritmos y Costes:

Constructor de Product
El constructor realiza validaciones sobre los datos de entrada:

1. ID: Comprueba que no sea negativo.
2. Nombre: Verifica que no sea nulo ni vacío.
3. Precio: Valida que no sea negativo.

El coste de estas validaciones es **O(1)**, dado que se trata de operaciones simples.

Métodos de Acceso (getters)

- `getName`, `getPrice` y `getId` permiten acceder a los atributos del producto.
- El coste de estas operaciones es **O(1)**, ya que únicamente devuelven el valor almacenado.

Método de Modificación (`setPrice`)

- Actualiza el precio de un producto, previa validación de que el nuevo precio no sea negativo.
- El coste es **O(1)**, pues la validación y la asignación son operaciones constantes.

Representación como Cadena (`toString`)

- Genera una descripción legible del producto, incluyendo su ID, nombre y precio.
- El coste depende de la longitud del nombre del producto y del formato del string. En promedio, es **O(n)**, donde *n* es la longitud del nombre.

4.7. Clase Shelf

La clase `Shelf` representa una estantería que contiene productos identificados por sus IDs, siendo capaz de gestionar dinámicamente su contenido. Utiliza una lista de enteros para almacenar los IDs de los productos en la estantería.

Estructura de datos utilizada:

Lista (`List<Integer>`): Utilizada para almacenar los IDs de los productos en la estantería. La elección de una lista permite mantener el orden de los productos y acceder a ellos de forma eficiente, aunque la complejidad de las operaciones depende de si los productos están o no duplicados.

Atributos:

- `name`: Representado como un `String`, almacena el nombre de la estantería. Se valida para garantizar que no sea nulo ni vacío en el constructor.
- `products`: Representado como un `List<Integer>`, almacena los IDs de los productos que están en la estantería. La lista permite gestionar dinámicamente los productos, añadiendo o eliminando elementos según sea necesario.

Algoritmos y Costes:

Constructor de Shelf
El constructor inicializa una nueva estantería, validando los datos de entrada:

- El nombre debe ser válido, es decir, no puede ser nulo ni vacío.
- Los IDs de los productos deben ser valores válidos, es decir, no deben ser negativos ni nulos.
- Si la lista de productos no es nula, se añaden los productos a la estantería evitando duplicados.
- El coste es **O(n)**, donde `n` es el número de productos, ya que debe verificar cada producto para asegurarse de que no sea nulo ni negativo.

Métodos de acceso (getters):

    - `getName()`: Devuelve el nombre de la estantería.
    - `getProducts()`: Devuelve una copia de la lista de productos, evitando modificaciones externas.
- Coste:
    - **O(1)** para `getName()`, ya que simplemente devuelve el valor almacenado.
    - **O(n)** para `getProducts()`, ya que se crea una nueva lista para evitar la modificación directa de la lista interna.

Método de adición de producto (`addProduct`):
Añade un producto a la estantería si no está duplicado. Verifica que el ID del producto no sea negativo antes de agregarlo a la lista.

- Coste **O(n)** en el peor de los casos, debido a la operación `contains` para verificar si el producto ya está en la lista. Si el producto no está presente, el coste de la inserción es **O(1)**.

Método de eliminación de producto (`removeProduct`):
Elimina un producto de la estantería si está presente. Se realiza una verificación de que el ID del producto no sea nulo.

- Coste **O(n)** en el peor de los casos, ya que se debe buscar el producto en la lista antes de eliminarlo.

Método de limpieza (`clearShelf`):
Elimina todos los productos de la estantería.

- Coste **O(n)**, donde `n` es el número de productos, ya que se recorre la lista para eliminar todos los productos.

Método de obtención del tamaño (`size`):
Devuelve el número de productos en la estantería.

- Coste **O(1)**, ya que el tamaño de la lista se mantiene actualizado y no requiere cálculos adicionales.

Representación como Cadena (`toString`):
Genera una representación legible de la estantería, incluyendo su nombre y la lista de productos.

- Coste **O(n)**, donde `n` es el número de productos, debido a la concatenación de los productos en la cadena.