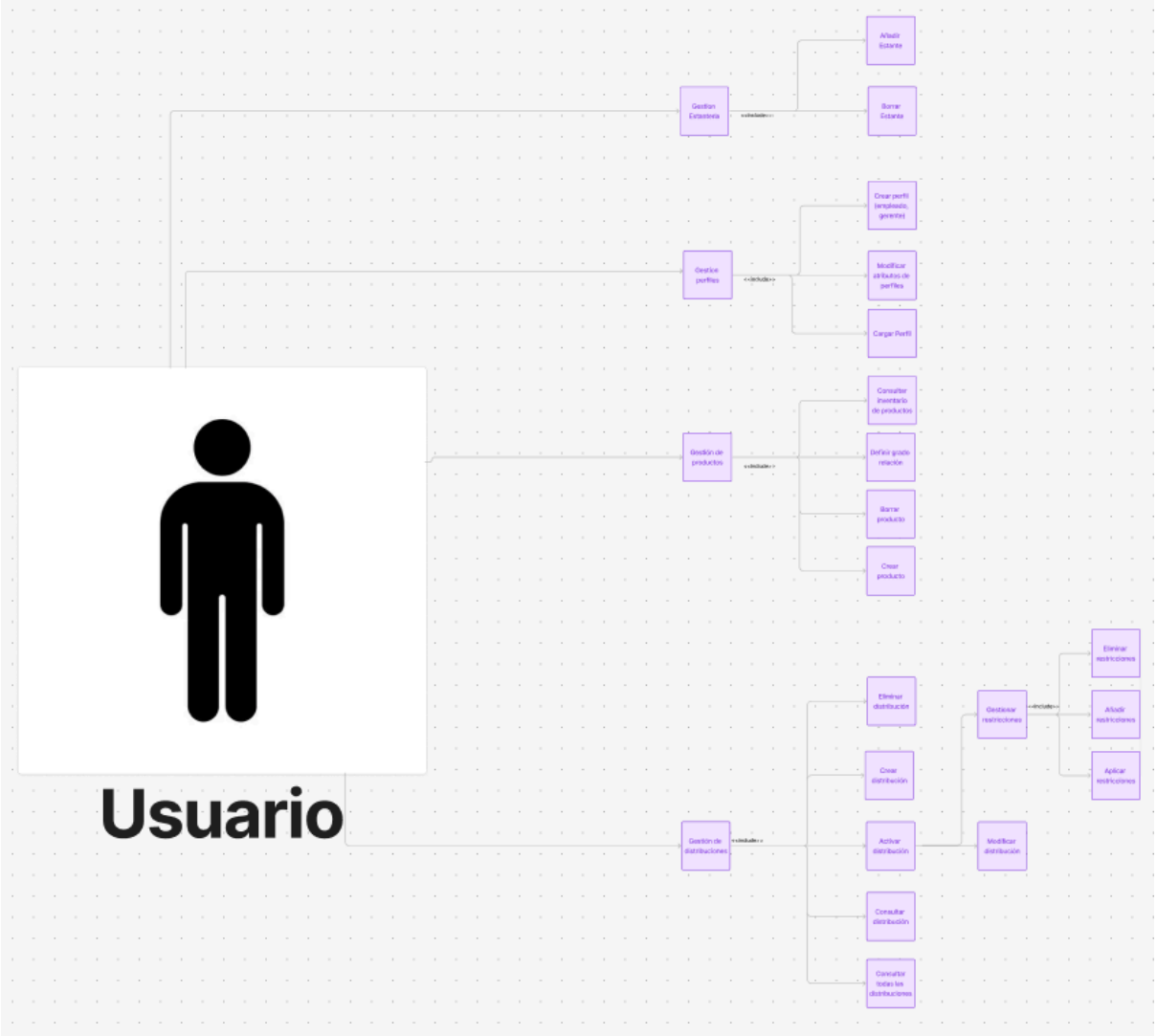
Distribución de productos

Identificador de equipo: 11.2
Joan Vila Orus: joan.vila.orus@estudiantat.upc.edu
Álvaro Monclús: alvaro.monclus@estudiantat.upc.edu
David Castro: david.castro.paniello@estudiantat.upc.edu
Natalia Yike Wang Yang: natalia.yike.wang@estudiantat.upc.edu

Versión 1.1

1ª Entrega

1. Diagrama de casos de Uso



Están los esquemas aquí y en el directorio DOCS.

1.1 Descripción Diagrama de casos de Uso

Caso de uso #1 Añadir estante:

Actor principal: El gerente
Detonante: Se quiere extender el número de estantes a la estantería del supermercado
Escenario principal:

1) El usuario proporciona el número de estantes que quiere añadir
2) El sistema añade el número de estantes

Extensiones:

1a) Se ha indicado un entero no positivo, el sistema indica el error y no modifica los estantes

Caso de uso #2 Borrar estante:

Actor principal: El gerente
Detonante: Se quiere borrar estantes a la estantería del supermercado
Escenario principal:

1) El usuario proporciona el número de estantes que quiere borrar
2) El sistema borra esa el número indicado de estantes

Extensiones:

1a) El usuario intenta eliminar más estantes de las que se pueden eliminar, la estantería siempre tiene un mínimo de 1 estante (número de estantes que quiere borrar >= número de estantes actuales): El sistema avisa del error y no modifica los estantes, espera un nuevo comando.

Caso de uso #3 Crear Producto:

Actor principal: Gerente
Detonante: El gerente desea añadir un producto nuevo al inventario.
Escenario principal:

1) El gerente proporciona el nombre del producto y la información necesaria que le pide el sistema.
2) Por cada producto ya existente el sistema pide al usuario que indique un grado de relación con el nuevo producto que se desea añadir.
3) El sistema añade el producto y guarda también todas las relaciones. Además muestra la información correspondiente del nuevo producto.

Extensiones:

1a) El producto que se desea añadir ya existe, el sistema informa del error y no añade el producto, espera un nuevo comando.

Caso de uso #4 Definir grado de relación:

Actor principal: Gerente
Precondición: Los 2 productos que ingresará el gerente están ya creados.
Detonante: El gerente desea definir una relación entre 2 productos.
Escenario principal:

1) El gerente proporciona el nombre de los 2 productos y el grado de relación que desee.
2) El sistema asigna el nuevo valor de la relación entre los 2 productos.

Extensiones:

1 a) El valor de la relación no está entre 0 y 1:El sistema avisa del error y no se hace nada.

1 b) Los dos productos son el mismo producto: El sistema avisa del error y se vuelve a pedir el grado de relación hasta obtener un valor válido.

Caso de uso #5 Consultar inventario de productos:

Detonante: El usuario quiere consultar los productos ya registrados en el sistema
Escenario principal:

1) El usuario indica que quiere consultar el inventario de productos.
2) El sistema muestra todos los productos registrados en el inventario con la información de cada uno.

Caso de uso #6 Borrar producto:

Actor principal: Gerente
Detonante: El gerente desea eliminar un producto.
Escenario principal:

3) El gerente proporciona el nombre del producto.
4) El sistema elimina el producto.

Extensiones:

1a) El producto no existe, el sistema indica el error y espera un nuevo comando.

Caso de uso #7 Consultar todas las distribuciones:

Detonante: El usuario quiere consultar todas las distribuciones ya registrados en el sistema
Escenario principal:

5) El usuario indica que quiere consultar todas las distribuciones.
6) El sistema muestra todos todas las distribuciones registradas en el sistema, independientemente de si hay alguna distribución y si están activas o no. Muestra la distribución activa en caso de que haya y seguidamente una lista de las distribuciones inactivas. Para cada distribución indica el ID identificador de la distribución y los productos correspondientes a cada estante según el número de Estantes que la estantería dispone en ese momento.

Caso de uso #8 Crear distribución:

Actor principal: El gerente
Precondición: Se ha configurado una estantería y hay un mínimo de 2 productos creados en el sistema.
Detonante: El usuario quiere crear una nueva distribución de la estantería de los productos registrados en el sistema
Escenario principal:

1) El usuario elige el algoritmo de cálculo entre 2 opciones: de “força bruta” o algoritmo voraz) i un algoritmo de aproximación.
2) El usuario indica el ID para identificar la nueva distribución.
3) El sistema calcula la distribución con el algoritmo elegido maximizando las relaciones entre productos y la muestra.
4) El sistema guarda la nueva distribución con el ID correspondiente.

Extensiones:

1a) El ID que se ha indicado ya existe, el sistema indica el error y espera un nuevo comando.

Caso de uso #9 Modificar distribución:

Actor principal: El gerente
Precondición: Hay una distribución activa en el sistema
Detonante: El usuario desea cambiar la posición de dos productos.
Escenario principal:

1) El usuario indica el identificador de los dos productos.
2) El sistema intercambia la posición de los dos productos indicados en la distribución activa.
1) El sistema muestra la distribución modificada de los productos

Extensiones:

1 a) Si se introduce algún ID de producto inexistente en el sistema se avisa del error y el sistema no hace nada.

1 b) Si se ha introducido un ID de algún producto que ya está restringido por alguna restricción que se está aplicando en la distribución activa el sistema avisa del error y espera un nuevo comando.

Caso de uso #10 Eliminar distribución:

Actor principal: El gerente
Precondición: La distribución que se desea eliminar existe en el sistema
Detonante: El usuario quiere eliminar la distribución que se ha configurado
Escenario principal:

1) El usuario indica el ID de la distribución quiere borrar
2) El sistema elimina la distribución que se ha configurado

Extensiones:

1 a) Se ha indicado un ID inexistente de distribución, el sistema avisa del error y espera un nuevo comando.

Caso de uso #11 Activar distribución:

Actor principal: El gerente
Precondición: Hay mínimo una distribución guardada
Detonante: El gerente desea cambiar o poner una distribución activa, para poder modificar o cambiar las posiciones de algunos productos en esa distribución.
Escenario principal:

1) El usuario indica el ID de la distribución que quiere activar.
2) El sistema activa dicha distribución.

Extensiones:

1 a) Se ha indicado un ID inexistente de distribución, el sistema avisa del error y espera un nuevo comando.

Caso de uso #12 Consultar distribución:

Actor principal: El gerente
Precondición: Hay mínimo una distribución registrada en el sistema
Detonante: El gerente desea consultar las posiciones de la distribución en la estantería
Escenario principal:

1) El usuario indica el ID de la distribución que desea ver en la estantería
2) El sistema muestra la distribución en función del número de estantes en ese momento.

Extensiones:

1 a) Se ha indicado un ID inexistente de distribución, el sistema avisa del error y espera un nuevo comando.

Caso de uso #13 Añadir restricción:

Actor principal: El gerente
Precondición: Hay al menos dos productos creados en el sistema y una distribución activa.
Detonante: El usuario quiere añadir restricciones a la distribución activa.
Escenario principal:

1) El gerente indica el ID de producto y la posición en la que desea colocar el producto.
2) El sistema guarda la restricción.

Extensiones:

1 a) Introduce una restricción invalida. El producto o la posición que se han indicado contradicen alguna restricción que tiene aplicada la distribución activa. El sistema avisa del error y espera un nuevo comando.

2 a) Introduce una restricción invalida. La posición indicada no es válida, supera el número de posiciones que ocupan los productos del inventario. El sistema avisa del error y espera un nuevo comando.

3 a) Introduce un ID de producto inexistente. La posición indicada no es válida, supera el número de posiciones que ocupan los productos del inventario. El sistema avisa del error y espera un nuevo comando.

Caso de uso #14 Eliminar restricción:

Actor principal: El gerente
Precondición: Hay una distribución activa.
Detonante: El usuario quiere eliminar alguna restricción de la distribución activa.
Escenario principal:

1) El gerente indica el ID de producto correspondiente a la restricción que desea eliminar.
2) El sistema elimina la restricción.

Extensiones:

1 a) Introduce un ID de producto que no tiene ninguna restricción. El sistema avisa del error y espera un nuevo comando.

2 a) Introduce un ID de producto inexistente. El sistema avisa del error y espera un nuevo comando.

Caso de uso #15 Aplicar Restricciones:

Actor principal: El gerente
Precondición: Hay una distribución activa
Detonante: El gerente quiere aplicar las restricciones añadidas a esta distribución
Escenario principal:

1) El gerente indica al sistema que quiere activar todas las restricciones de la distribución activa.
2) El sistema activa todas las restricciones a la distribución activa.

(OPCIONALES)
Caso uso #2 - Log in
Actor: Usuario registrado
Precondición: El usuario está registrado y no se ha loggeado
Detonante: El usuario quiere loggearse
Escenario Principal:

1) El usuario proporciona username y password
2) El sistema valida valores y proporciona acceso

Extensiones:

1a) El usuario no recuerda password: el sistema comienza el caso de uso – cambio de password (gerente)

2a) No existe un usuario con esas credenciales: el sistema avisa del error y vuelve al paso 1

Alta de Usuario
Actor: Gerente
Comportamiento:

El gerente elige hacer un alta, ha de entrar el nombre y apellidos del usuario, el código de usuario (username), la contraseña (password) –dos veces– y el tipo de usuario –que se escoge de una lista. El sistema valida valores y coherencia de los datos, y los registra.

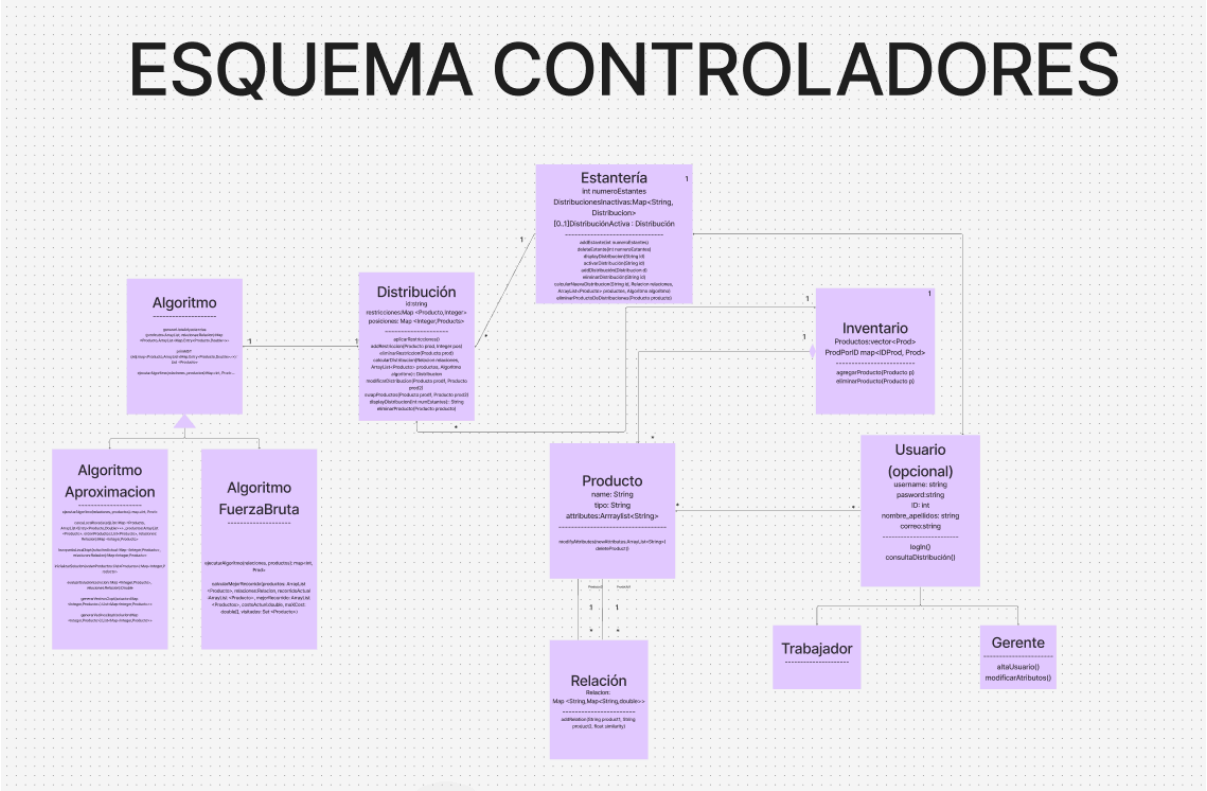
Errores posibles y cursos alternativos:
Este código de usuario ya existe: cambiarlo o abandonar
Las dos contraseñas no coinciden: volver a introducirlas

Modificar atributos perfiles
Actor: Gerente
Precondición: El usuario y gerente está registrado
Detonante: El usuario quiere modificar algún atributo
Escenario Principal:

1) El gerente proporciona username y password
2) El gerente proporciona el username que quiere modificar
3) El sistema muestra los atributos del perfil
4) El gerente modifica los valores.
3) El sistema valida valores y proporciona acceso

2. Diagrama del modelo conceptual

2.1 Diseño



Están los esquemas aquí y en el directorio DOCS.

2.2 Descripción de las clases

Nombre de la clase: PRODUCTO

Descripción breve de la clase: Producto de venta que queremos añadir en las posibles soluciones.

Multiplicidad: 1 por cada producto que se quiera registrar en el sistema.

Descripción de los atributos:
– Nombre: nombre del producto
– Tipo: Categoría al que pertenece el producto
– Atributos: Atributos del producto

Descripción de las relaciones:
– Relación de asociación con la clase "Relación": Todo producto tiene una relación con cada uno del resto de los productos.
– Relación de asociación con la clase "Inventario": Todo producto está incluido en el inventario del sistema.
– Relación de asociación con la clase "Usuario": Un producto puede ser registrado como una compra por un usuario.

Métodos:
modifyAttributes(newAttributes:ArrayList<String>):
Body: Modifica los atributos de dicho producto a los indicados en el parámetro.

deleteProduct():
Body: Elimina dicho producto del sistema.

Nombre de la clase: ESTANTERÍA

Descripción breve de la clase: Clase que representa la estantería circular del supermercado.

Multiplicidad: Sólo hay 1 estantería para todo el sistema.

Descripción de los atributos:
– Número de estantes**:** Estantes físicos que tiene la estantería, mínimo tiene que tener siempre 1 estante
– Distribución activa**:** Distribución que tiene la estantería en ese momento

Descripción de las relaciones:
– Relación de asociación con la clase "Distribución": Una estantería tiene mínimo 2 distribuciones posibles para colocar todos los productos en la estantería: aleatorio(Fuerza Bruta) y optimizado(Aproximación).

Métodos:

addEstante(int numeroEstantes):

Precondición: *numeroEstantes* es un número entero positivo.
Body: Se añade ese número de estantes en la estantería.

deleteEstante(int numeroEstantes):

Excepciones: *numeroEstantes* es mayor o igual que el número actual de estantes.
Body: Se elimina ese número de estantes de la estantería.

displayDistribucion(String id):

Excepciones: No existe una distribución con el identificador pasado como parámetro en la estantería.
Body: El sistema muestra dicha distribución adaptándose a las dimensiones de la estantería.

activarDistribución(String id):

Precondición: Existe una distribución con el identificador pasado como parámetro.
Body: Tal distribución pasa a ser la distribución activa de la estantería, para que el usuario pueda posteriormente gestionar las posiciones de los productos de esa misma distribución, independientemente de si ya había una distribución activa o no.

addDistribución(Distribucion d):

Precondición: No existe una distribución con el identificador pasado como parámetro.
Body: Se añade en el conjunto de distribuciones inactivas de la estantería.

eliminarDistribución(String id):

Precondición: Existe una distribución con el identificador pasado como parámetro.
Body: Se elimina del conjunto de distribuciones inactivas de la estantería, si correspondía a la distribución activa de la estantería, la estantería ya no tiene ninguna distribución activa.

calcularNuevaDistribucion(String id, Relacion relaciones, ArrayList<Producto> productos, Algoritmo algoritmo)

Precondición: Conjunto de productos contiene todos los productos del inventario, hay un mínimo de dos productos en el inventario, relaciones contiene las relaciones de todas las relaciones entre dos productos diferentes, algoritmo es un tipo de algoritmo en concreto.
Body: Se invoca al método **calcularDistribucion** de la clase Distribución, a continuación el sistema invoca al método **addDistribución(Distribucion d)** siendo d la nueva distribución calculada.

eliminarProductoDeDistribuciones(Producto producto):

Body: Se borra el producto de todas las distribuciones que se han registrado en la estantería invocando al método eliminarProducto de la clase distribución.

Nombre de la clase: DISTRIBUCIÓN

Descripción breve de la clase: Solución de las posiciones de todos los productos registrados en el sistema para colocarlos en la estantería

Multiplicidad: Puede haber mínimo 2: una aleatoria y otra optimizada en función de la relación de similitud entre los productos.

Descripción de los atributos:
– ID**:** Número de identificador único para cada distribución

Descripción de las relaciones:
– Relación de asociación con la clase "Inventario": Toda distribución debe incluir todos los productos del inventario.
– Relación de asociación con la clase "Estantería": Toda distribución debe adaptarse a las dimensiones de la estantería, específicamente según el número de estantes.
– Relación de asociación con la clase "Algoritmo": Toda distribución se ha creado a través del cálculo de un algoritmo en concreto, este puede ser de 2 formas: aleatorio(Fuerza Bruta) y optimizado(Aproximación)

Métodos:
aplicarRestricciones():
Precondición: Hay una distribución activa.
Body: Se aplican todas las restricciones que se han añadido para la distribución activa, para esta funcionalidad el sistema invoca al método **swapProductos** de la misma clase, donde pasa por parámetro el producto de la restricción y el producto que se encuentra en la posición de la restricción.

addRestriccion(Producto prod, Integer pos):
Precondición: Hay una distribución activa.
Excepciones: Hay alguna restricción con ese producto o esa posición de la distribución o la posición indicada es una posición inválida de la distribución.
Body: Se añade una nueva restricción para que ese producto esté en esta posición en la distribución activa.

eliminarRestriccion(Producto prod):
Precondición: Hay una distribución activa y esta distribución tiene una restricción para ese producto.
Body: Se elimina la restricción de producto de la distribución activa.

calcularDistribucion(Relacion relaciones, ArrayList<Producto> productos, Algoritmo algoritmo):: Distribucion:
Body: Se calcula una nueva distribución de los productos y relaciones pasados como parámetros en función del algoritmo pasado como parámetro.

modificarDistribucion(Producto prod1, Producto prod2)
Precondición: Hay una distribución activa.

Excepciones: Hay alguna restricción con ninguno de los productos pasados como parámetro.
Body: El sistema invoca al método **swapProductos** de la misma clase, donde pasa por parámetro los dos productos.

swapProductos(Producto prod1, Producto prod2):
Excepciones: Los dos productos son el mismo producto, los dos productos no se encuentran en la distribución.
Body: Se intercambian las posiciones de los dos productos en la distribución invocando al método **swapProductos** de la misma clase.

displayDistribucion(int numEstantes): String
Body: Se devuelve un String con los productos que hay en cada estante en el orden correspondiente adaptándose a las dimensiones del número de estantes pasados por parámetro.

eliminarProducto(Producto producto)
Excepciones: El producto no se encuentra en dicha distribución.
Body: Se elimina dicho producto de dicha distribución, se elimina las restricciones (si hay) de ese producto en dicha distribución, y se vuelve a ajustar la nuevo distribución de los productos.

Nombre de la clase: RELACIÓN

Descripción breve de la clase: Relación existente entre dos productos diferentes.

Multiplicidad: 1 relación por cada 2 productos diferentes existentes en el sistema.

Descripción de los atributos:
– Similitud : Define el grado de similitud del primer producto con el segundo producto y viceversa, es una relación bidireccional. Se representa con un número real de rango [0, 1], siendo 1 el máximo grado de relación entre los dos productos.

Descripción de las relaciones:
– Relación de asociación con la clase "Producto": Cada producto tiene una única relación con el resto de productos pero nunca consigo misma, además la relación es bidireccional.

Métodos:
addRelation(String product1, String product2, float similarity):
Precondición: *similarity* es un número real entre [0,1].
Excepciones: Los dos productos son el mismo producto.
Body: Se asigna el parámetro *similarity* como el grado de relación entre los dos productos, la relación es bidireccional.

Nombre de la clase: INVENTARIO

Descripción breve de la clase: Conjunto que incluye todos los productos.

Multiplicidad: Sólo hay 1 inventario en todo el sistema. (Singleton)

Descripción de los atributos:
– Productos **:** Contiene todos los productos que se han registrado en el sistema.
– ProductosPorID **:** Contiene todos los productos que se han registrado en el sistema ordenados por ID del producto para optimizar la búsqueda.

Descripción de las relaciones:
– Relación de asociación con la clase "Distribución": Toda distribución debe incluir todos los productos del inventario.
– Relación de asociación con la clase "Producto": El inventario contiene cada producto registrado en el sistema.

Métodos:
agregarProducto(Producto p):
Excepciones: Ya existe un producto en el inventario con mismo ID.
Body: Se añade dicho producto en el inventario.

eliminarProducto(Producto p):
Excepciones: No existe el producto en el inventario.
Body: Se borra dicho producto en el inventario y se borra de todas las distribuciones invocando al método **eliminarProductoDeDistribuciones()** de la clase Estantería.

Nombre de la clase: ALGORITMO

Descripción breve de la clase: Método en el que se calcula una distribución de productos, puede ser a fuerza bruta o optimizado.

Multiplicidad: Un algoritmo corresponde al método en el que se ha calculado una distribución.

Descripción de las relaciones:
– Relación de asociación con la clase "Distribución": Toda distribución se basa en un algoritmo en concreto.

Métodos:
ejecutarAlgoritmo(relaciones, productos):: Map<int, Prod>
Precondición: el conjunto de productos contiene más de 1 producto y contiene todos los productos registrados en el inventario, relaciones contiene las relaciones por cada pareja de productos diferentes del inventario.
Body: Se calcula una distribución en función del tipo de algoritmo al que pertenece: Aproximación o Fuerza Bruta.

generarListaAdyaciencias(productos,relaciones):Map<Producto,ArrayList<Map.Entry<Producto,Double>>>

Body: Genera una lista de adyacencia del grafo que tiene como nodos los productos del inventario y como aristas las relaciones entre ellos. Modificamos el peso de las relaciones para que en las aristas se vea multiplicado por -1 y sumado 1 para que vuelva a estar en el rango [0,1].

primMST (adj:map<Producto,ArrayList<Map.Entry<Producto,Double>>>) : List<Producto>

Body: Calcula el MST de la lista de adyacencia con el algoritmo de Prim y devuelve el orden en el que se visitaron los nodos con repetidos incluidos.

Nombre de la clase: ALGORITMO APROXIMACIÓN

Descripción breve de la clase: Implementación del algoritmo de aproximación para el cálculo de una distribución.

Métodos:

ejecutarAlgoritmo(relaciones, productos):: Map<int, Prod>

Precondición: el conjunto de productos contiene más de 1 producto y contiene todos los productos registrados en el inventario, relaciones contiene las relaciones por cada pareja de productos diferentes del inventario.

Body: Se calcula una distribución utilizando la implementación interior de aproximación

cercaLocalIterada(adjList, productos, ordenProductos, relaciones):Map<Integer,Producto>

Precondición: ordenProductos es una solución inicial de la distribución y no contiene productos repetidos.

Body: Realiza una búsqueda local iterada y devuelve la mejor solución encontrada. Utiliza la estrategia de intensificación y Hill-Climbing 2-opt y 3-opt.

busquedaLocal3opt(solucionActual, relaciones):Map<Integer,Producto>

Precondición: solucionActual es una solución inicial de la distribución y no contiene productos repetidos.

Body: Realiza una búsqueda local y devuelve la mejor solución encontrada. Utiliza Hill-Climbing 3-opt.

inicializarSolucion(ordenProductos):Map<Integer,Producto>

Body: Transforma una lista de productos a un mapa de integer, productos donde cada integer es la posición del producto en la lista.

evaluarSolucion(solucion, relaciones):Double

Precondición: relaciones contiene todas las relaciones sobre los productos de solucion.

Body: Acumula y devuelve las relaciones entre los elementos de la solución contiguos.

generarVecinos2opt(solucion):List<Map<Integer,Producto>>

Body: Genera vecinos de la solucion basándose en 2-opt swap.

generarVecinos3opt(solucion):List<Map<Integer,Producto>>

Body: Genera vecinos de la solucion basándose en 3-opt.

Nombre de la clase: ALGORITMO FUERZA BRUTA

Descripción breve de la clase: Implementación del algoritmo de fuerza bruta para el cálculo de una distribución.

Métodos:

ejecutarAlgoritmo(relaciones, productos):: Map<int, Prod>

Precondición: el conjunto de productos contiene más de 1 producto y contiene todos los productos registrados en el inventario, relaciones contiene las relaciones por cada pareja de productos diferentes del inventario.

Body: Se calcula una distribución utilizando la implementación interior de fuerza bruta.

calcularMejorRecorrido(productos relaciones, recorridoActual , mejorRecorrido:, costoActual, maXCost, visitados)

Body: recursivamente calcula y devuelve el orden de productos que maximiza la suma de relaciones contiguas. Computa todas las posibles soluciones, a no ser que calcule que no pueden ser la óptima ya, y devuelve la óptima.

(opcional)
Nombre de la clase: Usuario
Descripción breve de la clase: usuario registrado en el sistema
Multiplicidad: 1 único usuario por persona
Descripción de los atributos:
– username: nombre de usuario
– password: contraseña
– ID**:** número de identificador único para cada usuario
– nombre_apellidos: nombre completo del usuario
– correo: correo electrónico del usuario

Esta clase será implementada cuando se implemente la interfaz gráfica como una pantalla de login.

3. Relación de las clases implementadas por cada uno

Joan Vila	Alvaro Monclus	Natalia Yike	David castro
Inventario	Producto	Estanteria	Algoritmo
Algoritmo Aproximación	Distribucion	Relacion	Algoritmo FuerzaBruta
Juegos de Prueba	CtrlDomini y Drivers	Test JUnit	Ayuda en Algoritmo Aproximación

Cabe anotar que aunque cada clase haya tenido un autor principal, ha habido ayuda de todos los integrantes del equipo para resolver errores y optimizar.

4. Estructuras de datos y algoritmos utilizados

4.1 Estructuras de datos utilizados

Para la gestión de productos hay una clase Inventario que guarda los productos con un ArrayList que contiene todos productos registrados en el sistema. Por otra parte se ha implementado un Map que guarda los productos (value) en orden del ID del producto (key), en nuestro caso es el nombre del producto. Esto se debe a que productos tiene como clave externa el ID(nombre), de esta manera se gestiona correctamente y hay un único producto por ID.

En el caso de gestión de similitudes entre productos se ha utilizado un HashMap con estructura Map<String, Map<String, Float>>, donde cada String corresponde a un ID de producto y Float representa el grado de similitud entre los dos productos con un rango [0, 1] donde 1 representa la máxima relación entre dos productos. Esta relación entre productos es bidireccional, es decir, si se guarda un ID1 de producto con una relación con ID2 de 0.5, seguidamente el sistema registra para ID2 una relación con ID1 y 0.5. Por último un producto no tiene relaciones con él mismo, en nuestro caso inicializamos cada producto con relación 0 consigo mismo.

Para la implementación de los algoritmos, hemos optado por utilizar una estructura de datos *Map<Integer, Producto>* que nos permite devolver los resultados de forma organizada en un mapa. Esta estructura resulta particularmente conveniente para comprobar y almacenar los resultados, ya que nos permite acceder fácilmente a cada posición de la solución. Además, cuando deseamos añadir estantes, simplemente dividimos la solución en secciones y las invertimos en cada fila. Esta estrategia de división e inversión permite que múltiples estanterías mantengan una misma distribución, optimizando el proceso de almacenamiento y organización de productos.

En el algoritmo de **Fuerza Bruta**, usamos arreglos auxiliares que nos ayudan a registrar el mejor recorrido encontrado hasta el momento. Durante la ejecución, estos arreglos facilitan la comparación de rutas alternativas, de modo que el algoritmo siempre pueda optar por el recorrido de mayor beneficio acumulado. También empleamos un conjunto hash*set* de productos visitados, que garantiza que cada producto se visita solo una vez por recorrido. Esto no solo asegura la eficiencia del proceso, sino que evita ciclos repetitivos, manteniendo el algoritmo enfocado en encontrar soluciones óptimas.

Para el algoritmo de **Aproximación**, primero usamos un Map<Producto, ArrayList<Map.Entry<Producto, Double>>> para representar la lista de adyacencia de las relaciones si los productos fueran los nodos de un grafo. A partir de ahí, implementamos una lista de productos que utilizamos como base para aplicar el algoritmo de Prim, que genera un Árbol de Expansión Mínima (MST), en esta lista será el orden en el que se visitan los nodos en el MST con los nodos repetidos incluidos.
Luego usamos un mapa de integer y producto para representar soluciones parciales, y sets de visitados y listas de productos para eliminar los repetidos de una forma óptima.
En la cerca local iterada, también usamos el mapa para ir representando la solución y quedarnos con la óptima.

En las generaciones de vecinos para tanto 2-opt como 3-opt usamos una List<Map<Integer, Producto>> para representar los posibles vecinos y luego devolver el vecino que genera la solución óptima.

4.2 Algoritmos

Para simplificar hemos creado la clase abstracta Algoritmo donde cada uno de las algoritmos implementa su función “ejecutar algoritmo” en la cual cada uno realiza sus propias operaciones:

- El primer algoritmo que hemos utilizado es de Fuerza Bruta, un enfoque que explora todas las rutas posibles para asegurar que encuentra la mejor solución. Nuestra implementación se centra en maximizar el costo total del recorrido. Los pasos que sigue el algoritmo son los siguientes:
    1. Generar todas las rutas: Comienza con una lista de productos y busca recorrerlos todos de forma exhaustiva.
    2. Calcular el costo de cada ruta: Calcula la suma de relaciones definidas de la ruta.
    3. Evalúa qué ruta es la mejor y escoge el mejor resultado.
    4. Explora todas las rutas con las llamadas recursivas.

Ventajas y Desventajas:

Ventajas:

➔ Encuentra la MEJOR solución, ya que se prueban todas las rutas posibles.

➔ Es útil para conjuntos de datos pequeños, donde la precisión es crucial.

Desventajas:

➔ La complejidad del algoritmo es O(n!), muy costoso para grandes cantidades de productos

.

- El algoritmo que hemos utilizado para el **algoritmo de aproximación** está basado en una reducción al *Travelling Salesman Problem (TSP)* y se trata de una aproximación basada en el MST, la eliminación de repetidos y cerca local iterada con uso de Hill-Climbing 2-opt y 3-opt.
    1. Generamos lista de adyacencia: Generamos una lista de adyacencia del grafo que tiene como nodos los productos del inventario y como aristas las relaciones entre ellos. Modificamos el peso de las relaciones para que en las aristas se vea multiplicado por -1 y sumado 1 para que vuelva a estar en el rango [0,1]. De esta forma calculando el MST sobre la lista de adyacencia nos proporcionará en verdad el Spanning Tree que maximiza los pesos.
    2. Cálculo del MST: Utilizamos el algoritmo de Prim por simplicidad, y guardamos el orden en el que se recorren los nodos en su algoritmo incluyendo repetidos.
    3. Eliminación de repetidos: Este es un paso crucial en nuestro algoritmo, nuestra eliminación de repetidos es compleja y tiene carga computacional ya después solo se harían búsquedas de óptimos. Entonces hemos decidido des de cada nodo del ciclo euleriano ir recorriendo los demás nodos y eliminar los que nos encontremos por segunda vez, después cuando ya tenemos diferentes listas de productos sin repetir, hacemos búsqueda local

iterada en cada lista para encontrar una solución cercana a la óptima y aprovechar el tiempo computacional para mejorar nuestra solución inicial.

- Nuestra búsqueda local iterada con la estrategia de intensificación consiste en estos pasos, copiando la estructura dada en la documentación de la asignatura:
    1. Generar solución inicial: Partiremos de una de las listas de MST sin repetidos
    2. Empezamos un bucle del que saldremos cuando no hay habido mejora 10 veces.
    3. Hacemos búsqueda local sobre la solución actual, tanto con 3-opt como con 2-opt.
    4. Actualizamos la mejor solución.
    5. “Aceptamos” a nuestra mejor solución, ya que seguimos la estrategia de intensificación.
    6. Consideramos como nuestra solución perturbada a la mejor solución encontrada con el 2-opt
    7. Guardamos la solución perturbada como solución actual
    8. Repite el proceso.

- Nuestro proceso de búsqueda local consiste en Hill-Climbing, y hemos hecho una versión 3-opt y una versión 2-opt que usamos para perturbar las soluciones en la búsqueda local iterada. Sus pasos consisten en:
    1. Solución Inicial: empezamos des de una solución dada por el proceso de búsqueda local iterada.
    2. Mirar a tu alrededor: Observar las soluciones “vecinas” modificadas de la solución inicial.
    3. Elegir el mejor Camino: Comparar cual de las soluciones “vecinas” es la mejor.
    4. Repite el proceso.
    5. Alcanza el óptimo global: No se obtiene ninguna solución “vecina” mejor que la actual el algoritmo termina.

Ventajas y Desventajas del Algoritmo de Aproximación:

Ventajas:

- ➔ Algoritmo sencillo y rápido que encuentra soluciones bastante buenas en muchos casos.
- ➔ Como no es muy costoso, le aplicamos varias técnicas de búsquedas de mejores óptimos para obtener mejores resultados.
- ➔ No se puede quedar atrapado en soluciones que son buenas localmente pero no las mejores posibles (óptimos locales).
- ➔ Como hacemos búsquedas para varias versiones del MST sin repetidos, hay mucha más garantía de encontrar una solución más cercana a la óptima.

Desventajas:

- ➔ Sobrecosto computacional inicial, si quieres que sea extremadamente eficiente, se debería considerar no usar la cerca local iterada.
- ➔ Dependencia sobre el MST, siempre partimos de él para todas las soluciones, la calidad de nuestra solución depende de él.