DISTRIBUCIÓN DE PRODUCTOS EN UN SUPERMERCADO

PROYECTO DE PROGRAMACIÓN
FIB
16 de Diciembre de 2024

Equipo 43.4

- Nil Casas Duatis _____nil.casas.duatis
- Andreu Corden Moragrega _____andreu.corden

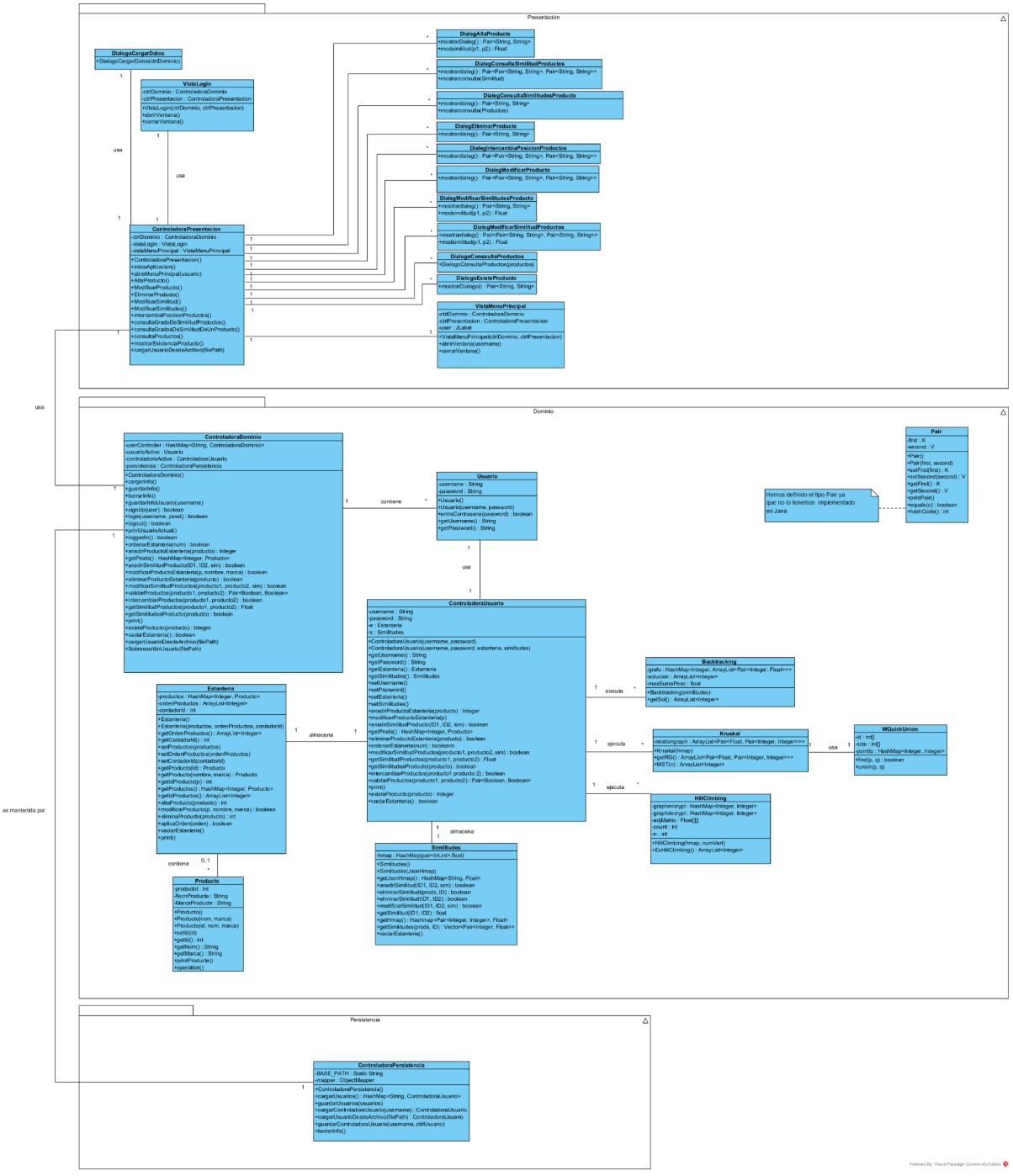
Versión 2.0

ÍNDICE

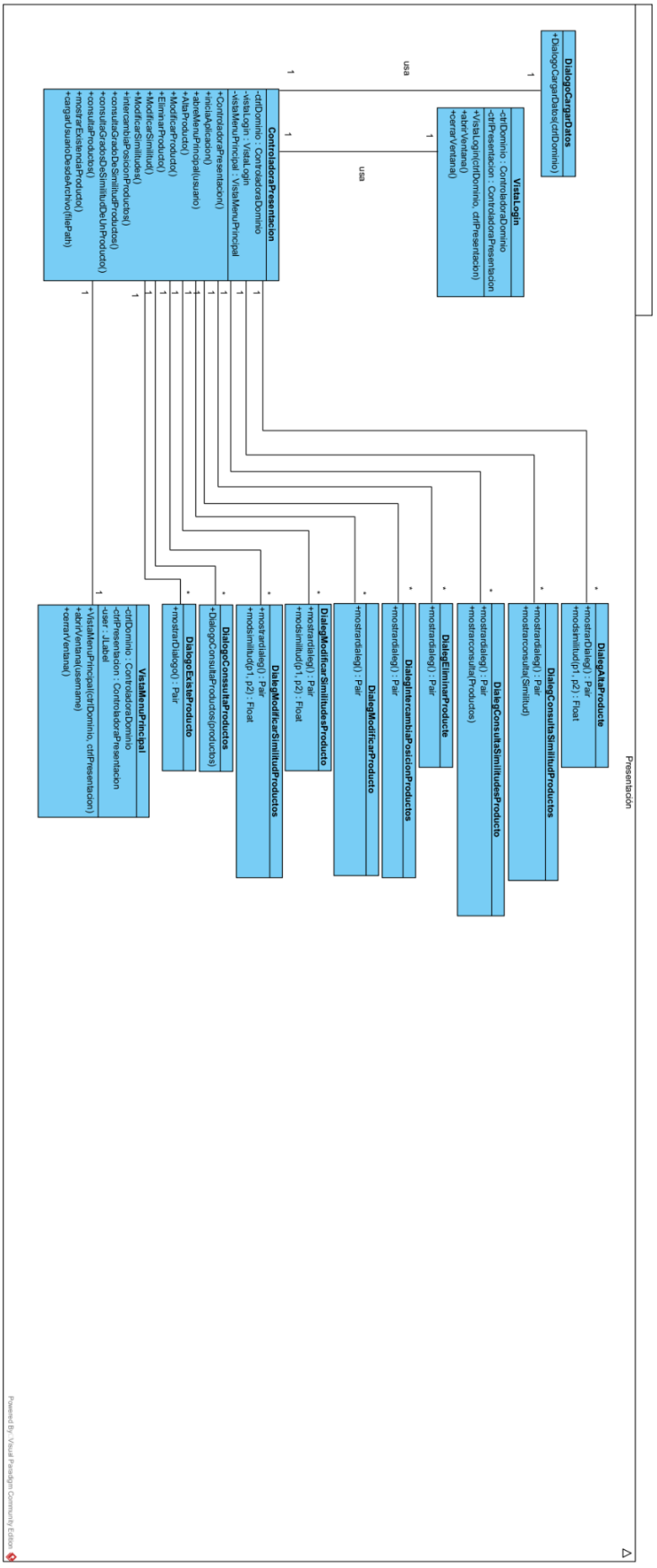
1.- Cambios respecto la 1ª Entrega

1.1.- Diagramas UML de las 3 capas

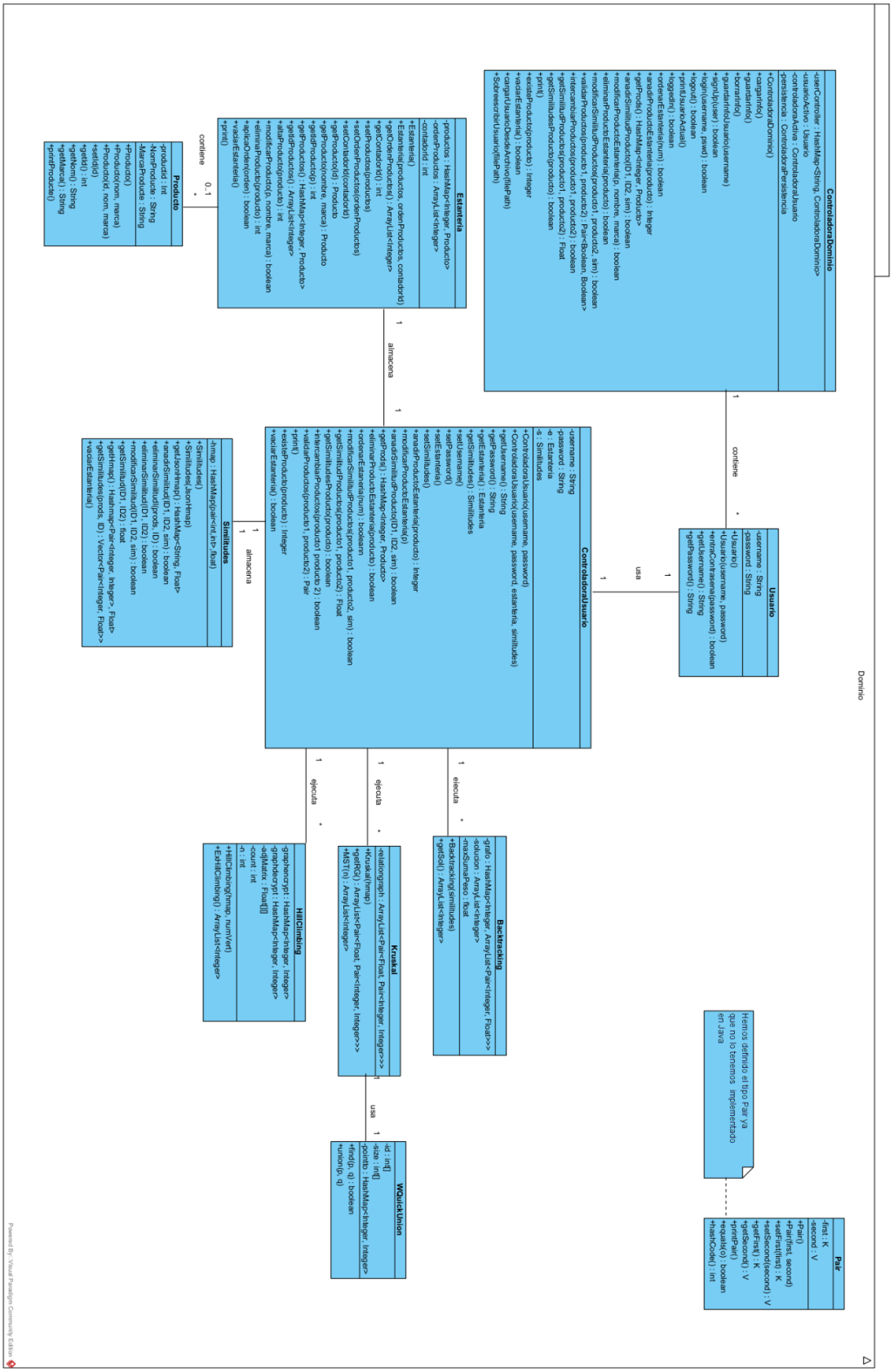
1.1.1.- Diagrama completo



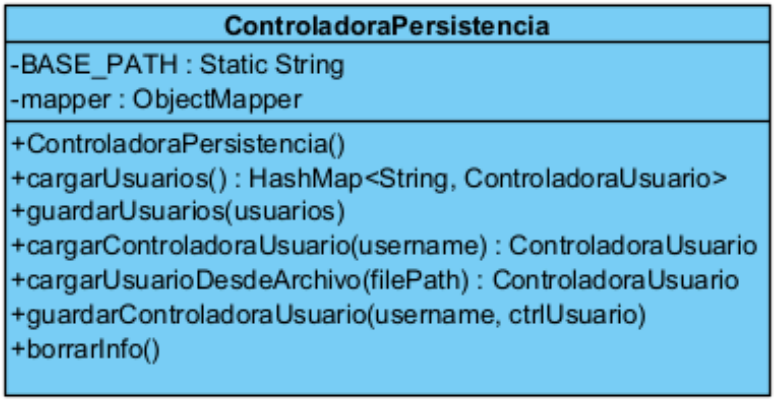
1.1.2.- Diagrama de la capa de presentación



1.1.3.- Diagrama de la capa de dominio



1.1.3.- Diagrama de la capa de persistencia



1.2.- Descripción de clases de cada capa

Estas descripciones són generales y cada cambio concreto se muestra en verde en el apartado 3.

1.2.1.- Descripción de la capa de presentación

Para la capa de presentación hemos delegado la responsabilidad a la controladora de presentación, que se encargará de ejecutar los casos de uso de la controladora del dominio según la información que el usuario indique a través de la interfaz.

Con la nueva capa de presentación hemos hecho que la controladora del dominio se encargue de mostrar por terminal la información a modo de debug.

1.2.2.- Descripción de la capa de dominio

En esta entrega lo primero que hemos hecho es renombrar las clases *“CjtUsuarios”* y *“ControladoraDominio”* a *“ControladoraDominio”* y *“ControladoraUsuario”* respectivamente.

Este cambio lo hemos hecho para poder gestionar la cohesión de las nuevas clases para gestionar la Presentación con “ControladoraPresentacion” y la persistencia con *“ControladoraPersistencia”.*

Hemos añadido el caso de uso de ModificarProducto, que nos permite cambiar su nombre y/o marca y no habíamos considerado en la primera entrega. También queremos recalcar que nuestro caso de uso “Consultar Estantería“ es “Consulta Productos” ya que hubo confusiones con la primera entrega.

En la “ControladoraDominio” hemos eliminado el atributo “Credenciales”, ya que para cada “ControladoraUsuario” almacenaremos el usuario y contraseña de este (Accesible desde UserController).

Este cambio ha sido para tener más cómodo la serialización de deserialización del sistema de ficheros en el formato que usamos en persistencia.

1.2.3.- Descripción de la capa de persistencia

Para la capa de persistencia hemos usado JACKSON, una librería de procesamiento de JSON. Para llevarlo a cabo, hemos añadido una nueva clase “Controladora Persistencia” que se encarga de leer y escribir los datos de otras ejecuciones para guardar el progreso de cada usuario. Para ello, hemos tenido que permitir que las clases del dominio que se instancian al iniciar una ejecución sean serializables, indicandolo con “implements serializable”.

En el caso de la clase similitudes, hemos tenido que modificarla con un atributo llamado jsonHmap definido como “private HashMap<String, Float> jsonHmap” para poder serializar y deserializar el atributo “HashMap<Pair<Integer, Integer>, Float> hmap”, ya que JSON nos daba problemas para este paso. Mantenemos información duplicada y entendemos que se podría cambiar similitudes para tener solo el atributo con String como key, pero al ser la antigua clase de David y ser 2 integrantes, hemos decidido centrarnos en lo demás por tema de tiempo.

También hemos añadido las funciones creadoras, getters y setters necesarias para que JSON pueda controlar esta información.

Para ello hemos implementado en la clase “ControladoraPersistencia” las siguientes funciones:

- cargarUsuarios(): HashMap <String, ControladoraUsuario>
- guardarUsuarios(usuarios)
- cargarControladoraUsuario(username):ControladoraUsuario
- cargarUsuarioDesdeArchivo(filePath):ControladoraUsuario
- guardarcontroladoraUsuario(username,ctrlUsuario)
- borrarInfo()

1.3.- Descripción de las estructuras de datos y algoritmos

En este apartado, respecto a la primera entrega, hemos discutido las decisiones tomadas en la elección de cada estructura de datos y algoritmo para justificar el uso de los atributos seleccionados.

1.4.- Modificaciones de los casos de uso

Primero de todo queremos indicar que el caso de uso de consultar la estantería lo hemos nombrado “consulta productos” ya que al ser un grupo de “3” solo teníamos una estantería. No estamos consultando los productos de una estantería en concreto sino que consultamos los productos de la única estantería que hay.

Aquí vienen nuestros cambios:

- Caso de uso de modificar producto añadido (no lo habíamos considerado en la primera entrega).

- También hemos añadido el caso de uso de cargar la información de un usuario desde un fichero en formato json. Este caso de uso se podrá ejecutar en la vista de iniciar sesión.

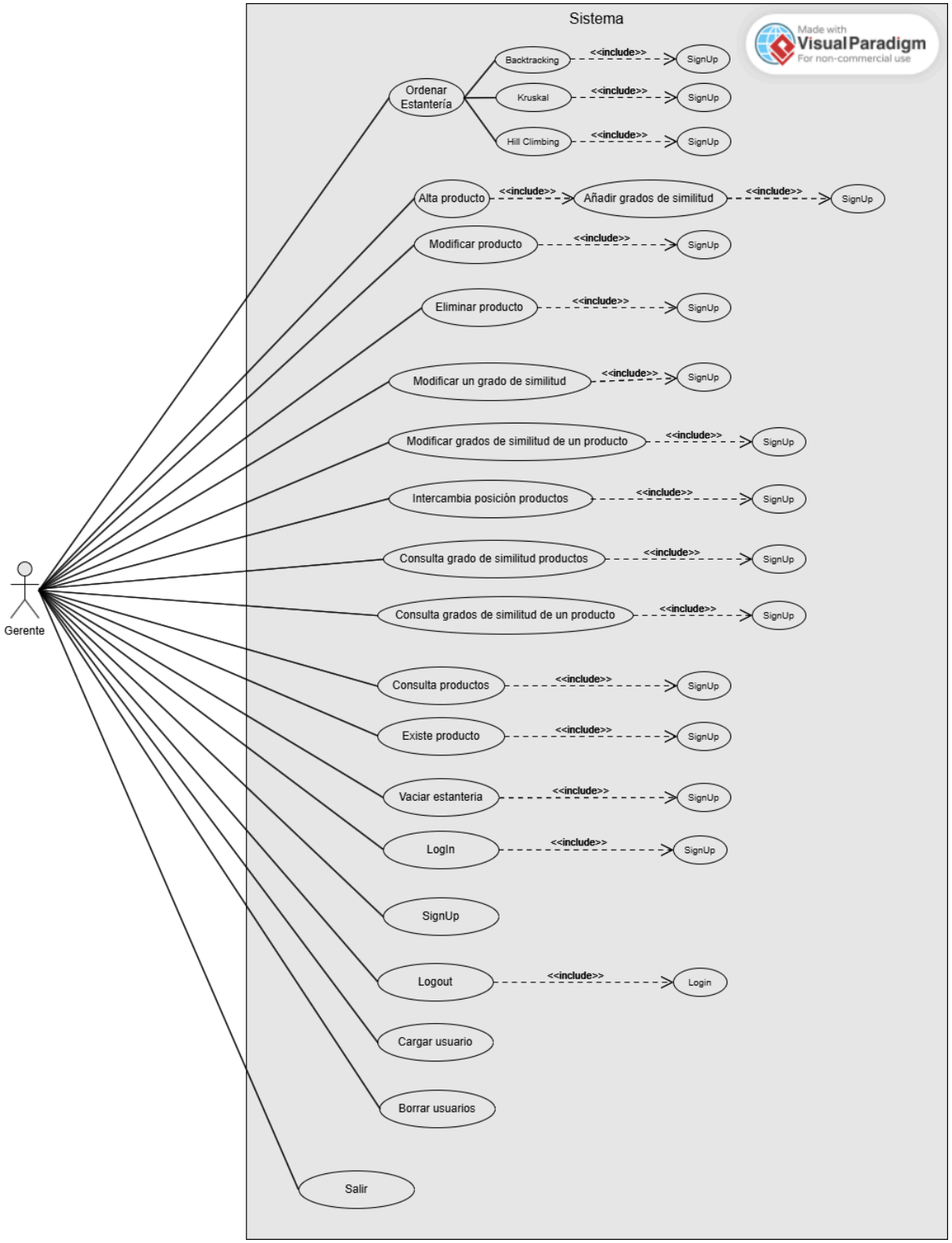
- Otro caso de uso nuevo es cargar los usuarios desde el directorio descrito por la capa de persistencia.

El diagrama de los casos de uso, entonces, ha sido modificado con los 3 casos de uso nuevos mencionados anteriormente (modificarProducto, cargarUsuario y borrarUsuarios) y hemos añadido sus descripciones en cada subapartado del apartado 2. También se ha modificado el sentido de los includes del diagrama, ya que eran incorrectos.

A continuación, tenemos el proyecto actualizado con los cambios mencionados:

2.- Casos de uso

2.1.- Diagrama



2.2.- Descripción

2.2.1.- Casos de uso obligatorios

• Nombre: Ordenar Estantería
• Actor: Gerente
• Comportamiento:
1) El gerente informa al sistema que quiere ordenar la estantería.
2) El sistema le muestra al gerente los 3 métodos de ordenación.
3) El gerente escoge uno de los métodos.
4) El sistema lo procesa y salta al caso de uso: *"Backtracking", "Kruskal" o "Hill Climbing"* dependiendo de la elección del gerente.

• Errores posibles y cursos alternativos:
3a) Si el gerente introduce un método distinto a los disponibles el sistema vuelve a mostrar los 3 métodos de ordenación.
4a) Si la estantería está vacía (y por lo tanto no hay productos) el sistema informa de la situación.

• Nombre: Backtracking
• Actor: Gerente
• Comportamiento:
1) El gerente informa al sistema que quiere ordenar la estantería mediante el algoritmo de Backtracking.
2) El sistema ordena la estantería mediante este método seleccionado y muestra el resultado.

• Errores posibles y cursos alternativos:
2a) Si la estantería está vacía (y por lo tanto no hay productos) el sistema informa de la situación.

• Nombre: Kruskal
• Actor: Gerente
• Comportamiento:
1) El gerente informa al sistema que quiere ordenar la estantería mediante el algoritmo de Kruskal.
2) El sistema ordena la estantería mediante este método seleccionado y muestra el resultado.

• Errores posibles y cursos alternativos:
2a) Si la estantería está vacía (y por lo tanto no hay productos) el sistema informa de la situación.

• Nombre: Hill Climbing
• Actor: Gerente
• Comportamiento:
1) El gerente informa al sistema que quiere ordenar la estantería mediante el algoritmo de Hill Climbing.
2) El sistema ordena la estantería mediante este método seleccionado y muestra el resultado.
• Errores posibles y cursos alternativos:
2a) Si la estantería está vacía (y por lo tanto no hay productos) el sistema informa de la situación.

• Nombre: Alta producto
• Actor: Gerente
• Comportamiento:
1) El gerente informa al sistema qué producto quiere añadir.
2) El sistema comprueba que el producto no exista y (en caso de que no exista) lo añade al sistema y salta al caso de uso: *"Añadir grados de similitud"*.
• Errores posibles y cursos alternativos:
2a) Si ya existía el producto el sistema informa del error.

• Nombre: Añadir grados de similitud
• Actor: Gerente
• Comportamiento:
1) El sistema presenta la lista de productos almacenados en el sistema.
2) El gerente introduce los grados de similitud de todos los productos disponibles.
3) El sistema almacena los grados de similitud proporcionados.
• Errores posibles y cursos alternativos:
2a) Si el grado de la similitud no tiene un formato correcto el sistema informa del error y vuelve a leer la información.

• Nombre: Modificar producto
• Actor: Gerente
• Comportamiento:
1) El gerente informa al sistema qué producto quiere modificar.
2) El sistema comprueba que el producto exista y pregunta el nuevo nombre y marca del producto
3) El gerente introduce el nuevo
4) El sistema modifica el producto según el nuevo nombre y marca que haya insertado el gerente
• Errores posibles y cursos alternativos:
2a) Si no existía el producto el sistema informa del error.
4a) Si los valores introducidos ya los tiene otro producto, el sistema informa del error

• Nombre: Eliminar producto
• Actor: Gerente
• Comportamiento:
1) El gerente informa al sistema qué producto quiere eliminar.
2) El sistema comprueba que el producto esté registrado y lo elimina.
3) El sistema elimina todas las similitudes que contengan este producto.
• Errores posibles y cursos alternativos:
2a) Si no existe el producto con el nombre indicado, el sistema informa del error.

• Nombre: Modificar un grado de similitud
• Actor: Gerente
• Comportamiento:
1) El gerente/sistema informa al sistema de qué productos quiere cambiar el grado de similitud y cuál será el nuevo grado.
2) El sistema comprueba que los dos productos estén registrados en el sistema y cambia el valor de grado de similitud.
• Errores posibles y cursos alternativos:
2a) Si no existe uno de los productos con el nombre indicado (o ambos), el sistema informa del error.
2b) Si el grado de la similitud no tiene un formato adecuado, el sistema informa del error y vuelve a leer la información.

• Nombre: Modificar grados de similitud de un producto
• Actor: Gerente
• Comportamiento:
1) El gerente informa al sistema de qué producto quiere cambiar todos sus grados de similitud y los nuevos valores de estos.
2) El sistema comprueba que el producto está registrado en el sistema y cambia todos sus grados de similitud si son adecuados ejecutando "*Modificar un grado de similitud*".
• Errores posibles y cursos alternativos:
2a) Si no existe el producto con el nombre indicado, el sistema informa del error.
2b) Si el valor de la similitud no es adecuado, el sistema informa del error y vuelve a leer la información.

• Nombre: Intercambia posición productos
• Actor: Gerente
• Comportamiento:
1) El gerente informa al sistema que quiere intercambiar la posición de dos productos en la estantería.
2) El sistema comprueba que los dos productos estén registrados en el sistema e intercambia su posición en la estantería.

• Errores posibles y cursos alternativos:
2a) Si no existe uno de los productos con el nombre indicado (o ambos), el sistema informa del error.
2b) Si los productos indicados son iguales el sistema informa del error.

• Nombre: Consulta grado de similitud productos
• Actor: Gerente
• Comportamiento:
1) El gerente informa al sistema que quiere conocer el grado de similitud que tienen dos productos determinados.
2) El sistema comprueba que los dos productos estén registrados en el sistema y muestra su grado de similitud.
• Errores posibles y cursos alternativos:
2a) Si no existe uno de los productos con el nombre indicado (o ambos), el sistema informa del error.

• Nombre: Consulta grados de similitud de un producto
• Actor: Gerente
• Comportamiento:
1) El gerente informa al sistema que quiere conocer el grado de similitud que tiene un producto con los demás.
2) El sistema comprueba que el producto esté registrado en el sistema y muestra una lista formada por tuplas compuestas de los nombres de los productos y sus grados de similitud.

• Errores posibles y cursos alternativos:
2a) Si no existe el producto con el nombre indicado, el sistema informa del error.

• Nombre: Consulta productos
• Actor: Gerente
• Comportamiento:
1) El gerente informa al sistema que quiere conocer todos los productos registrados en el sistema.
2) El sistema muestra una lista formada por los nombres de los productos.
• Errores posibles y cursos alternativo
2) Si no hay ningún producto registrado, el sistema muestra un mensaje de error indicándolo.

• Nombre: Existe producto
• Actor: Gerente
• Comportamiento:
1) El gerente introduce el nombre del producto en el sistema.
2) El sistema informa de la existencia del producto insertado.

• Nombre: Salir
• Actor: Gerente
• Comportamiento:
1) El gerente informa al sistema que quiere salir del programa.
2) El sistema cierra la aplicación.

• Nombre: Cargar Usuario Desde Archivo
• Actor: Gerente
• Comportamiento:
1) El gerente informa al sistema qué archivo quiere cargar.
2) El sistema inserta el nuevo usuario en la base de datos.
• Errores posibles y cursos alternativos:
2a) Si el usuario a cargar ya existe el sistema pregunta al gerente si quiere sobreescribirlo.
2b) Si el usuario a cargar no tiene un formato correcto o tiene datos corrompidos, el sistema informa del error.

2.2.2.- Casos de uso opcionales

• Nombre: Signup
• Actor: Gerente
• Comportamiento:
1) El gerente informa de que quiere crear un nuevo usuario en el sistema
2) El sistema pregunta por usuario y contraseña y añade el nuevo usuario al sistema.
• Errores posibles y cursos alternativo
2a) Si el usuario ya existe, el sistema muestra un mensaje de error.

• Nombre: Login
• Actor: Gerente
• Comportamiento:
1) El gerente informa de que quiere iniciar sesión en el sistema.
2) El sistema pregunta por el usuario.
3) El sistema pregunta por la contraseña.
4) El usuario introduce la contraseña
5) El sistema inicia sesión
• Errores posibles y cursos alternativo
2a) Si el usuario no existe el sistema envía un mensaje de error comunicandolo.
5a) Si la contraseña introducida no coincide el sistema informa del error.

• Nombre: Logout
• Actor: Gerente
• Comportamiento:
1) El gerente informa de que quiere cerrar su sesión en el sistema

2) El sistema cierra la sesión

• Errores posibles y cursos alternativo

2a) Si no hay ninguna sesión iniciada, el sistema informa del error.

• Nombre: Vaciar estantería
• Actor: Gerente
• Comportamiento:
1) El gerente informa de que quiere vaciar la estantería.
2) El sistema elimina los productos de la estantería, asimismo todas sus similitudes y las deja como en su estado inicial.

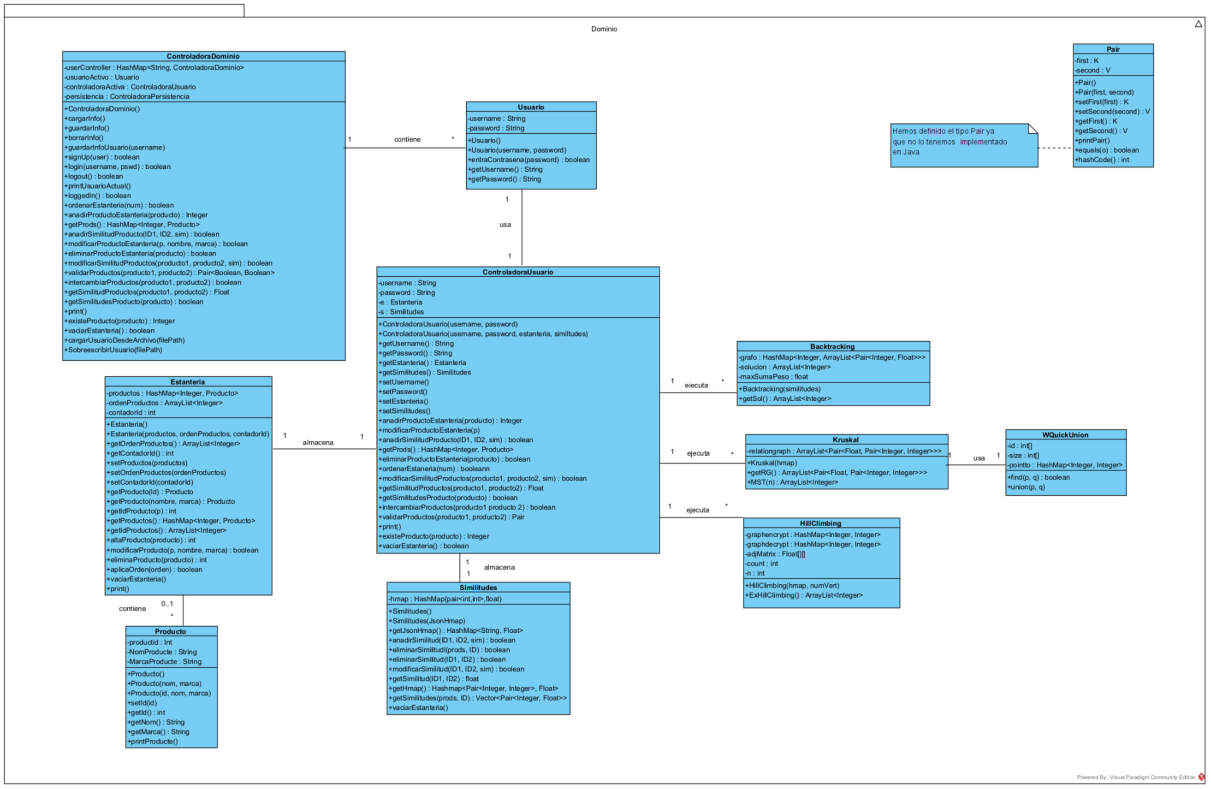
• Nombre: Borrar Usuarios
• Actor: Gerente
• Comportamiento:
1) El gerente indica al sistema que quiere eliminar todos los archivos guardados.
2) El sistema elimina los archivos .json de los que dispone.

• Errores posibles y cursos alternativos:

3.- Clases

3.1 Capa de Dominio

3.1.1.- Diagrama de clases de diseño del Dominio



3.1.2.- Breve descripción de las clases

3.1.2.1.- Pair

Nombre: Pair

- Breve descripción de la clase: Contiene una pareja de elementos
- Cardinalidad: .
- Descripción de los atributos:
  - *First:* Primer elemento de la pareja.
  - *Second:* Segundo elemento de la pareja.
- Descripción de los métodos:

  Pair() : Creadora de la clase, inicializa un Pair con sus valores a Null.

  Pair(first, second) : Creadora con parámetros inicializados. El primer elemento del Pair será first y el segundo second.

setFirst(first) : Modificadora del primer elemento del Pair. Asigna first al primer elemento del Pair y retorna este para la comprobación de errores fuera de la clase.

setSecond(second) : Modificadora del segundo elemento del Pair. Asigna second al segundo elemento del Pair y retorna este para la comprobación de errores fuera de la clase.

getFirst() : Consultora del primer elemento del Pair. La función devuelve este.

getSecond() : Consultora del segundo elemento del Pair. La función devuelve este.

printPair(): Escritora del Pair. Escribe los valores entre paréntesis y separados con una coma. En el caso de que algún valor sea Null, se imprime un espacio en el lugar del valor.

Inicializado: (2,0.3) Sin inicializar: (,)

equals(o) : Función que permite comparar dos Pairs para saber si su contenido es el mismo o distinto, retorna true y false respectivamente.

hashCode() : Función que permite usar la clase Pair en un HashMap, hace y retorna un código hash válido para ser usado en un HashMap.

3.1.2.2.- Producto

Nombre: Producto
• Breve descripción de la clase: Contiene toda la información relacionada con el producto.
• Cardinalidad: Existe un elemento por cada producto introducido al sistema.
• Descripción de los atributos:
- *Identificador:* Es el número identificador del producto.
- *Nombre:* Identifica el nombre del producto.
- *Marca:* Identifica de qué marca es el producto.

• Descripción de los métodos:
- *Producto(Identificador, Nombre, Marca):* Creadora de producto donde se asigna un identificador, un nombre y una marca al nuevo producto.
- *Producto(Nombre, Marca):* Creadora de un producto donde se asigna un nombre y una marca a un producto nuevo. Al identificador se le da un valor fijo que será sustituido.
- *setId(Identificador):* El valor del identificador del producto se convierte en el de id.
- *getid():* Devuelve el valor del identificador del producto.
- *getNom():* Devuelve el nombre del producto.
- *getMarca():* Devuelve la marca del producto.
- *printProducte():* Imprime el nombre y la marca del producto.

• Descripción de las relaciones:
  - *Contiene:* La relación indica en qué posición de la estantería está almacenado el producto.

3.1.2.3.- Estantería

• Nombre: Estanteria
• Breve descripción de la clase: Clase contenedora de productos. Almacena los productos en un orden determinado y le asigna un orden.
• Cardinalidad: * (una por usuario)
• Descripción de los atributos:
  - *productos*: Indica qué productos hay en ese momento en la estantería. En esta estructura de datos tenemos los productos indexados por su id en un diccionario, por lo que podemos acceder a ellos eficientemente. (no static)
  - *ordenProductos*: Indica qué productos hay en ese instante en la estantería, cada producto tiene una posición determinada, por lo que también están ordenados, al ejecutar el algoritmo se reordenará este mismo atributo. (no static)
  - *contadorId*: Indica el id que tendrá el siguiente producto a añadir a la estantería. (no static)
• Descripción de los métodos:
  - *Estanteria*: Función creadora, crea una estantería vacía.
  - *getIdProductos*: Función getter que devuelve un ArrayList<Integer> con los identificadores de los productos ordenados por el orden actual de la estantería.
  - *getProducto(id)*: Función getter que dado un identificador de un producto, devuelve la instancia de ese producto o null, si no existe
  - *getProducto(p):* Función getter que dado un Producto con nombre y marca pero identificador no válido devuelve ese producto pero ahora con un identificador válido.
  - *getIdProducto(p):* Función getter que dado un Producto con nombre y marca pero identificador no válido devuelve el id que tiene asignado en la estantería.
  - *getProductos()*: Función getter que devuelve un HashMap<Integer,Producto> donde se representan los Productos con Identificador como Key.
  - *getIdProductos()*: Función getter que se encarga de devolver una lista de identificadores de productos que representa el orden actual de la estantería.
  - *altaProducto(p):* Función setter que dado un Producto con nombre y marca pero identificador no válido lo inserta en la estantería asignando un id válido y devolviendo este id.
  - *eliminaProducto(p):* Función setter que dado un Producto con nombre y marca pero identificador no válido elimina el producto de la estantería y devuelve su id o -1 en caso que el producto no estuviese en la estantería.

- *aplicaOrden(orden):* Función setter que dado un vector de ids de Productos representando un orden, asigna ese orden a la estantería si es válido y devuelve un booleano indicando la correcta ejecución (o no) de esta asignación.
- *VaciarEstanteria():* Función que, tal y como su nombre indica, se encarga de eliminar todos los productos de la estantería.
- *print:* Función que muestra por el canal de salida estándar el estado actual de la estantería.

• Descripción de las relaciones:
- Relación *"contiene"* con la clase "Producto": Indica qué productos hay en la estantería.
- Relación *"almacena"* con la clase "ControladoraDominio", indica a qué instancia de la clase controladora del dominio pertenece esta estantería.

3.1.2.4.- Similitudes

Nombre: Similitudes
• Breve descripción de la clase: Contiene toda la información de las similitudes entre los productos de una estantería.
• Cardinalidad: Existe un elemento por cada usuario registrado.
• Descripción de los atributos:
- *Hmap:* Contenedor de cada similitud entre dos productos. La clave del mapa es un Pair de los Id's de los productos y como valor contiene la similitud entre estos. Hay similitudes duplicadas pero con las posiciones de los productos intercambiadas, se hizo de esta manera para facilitar los algoritmos de ordenación.

• Descripción de los métodos:

Similitudes() : Creadora de la clase. Inicializa hmap con un HashMap vacío.

anadirSimilitud(prod1, prod2, sim) : Añade la similitud de los productos introducidos al hmap. Para eso comprueba si ya existe esta y si no es el caso, añade dos elementos:

(<prod1, prod2>, sim) y (<prod2, prod1>, sim)

Retorna true si se ha añadido correctamente, false en caso contrario.

eliminarSimilitud(prods, prod1) : Elimina todas las similitudes de hmap que tengan como clave una combinación de prod1 y los productos de prods. Retorna true si se han eliminado correctamente, false en caso contrario.

eliminarSimilitud(prod1, prod2) : Elimina del hmap las claves <prod1, prod2> y <prod2, prod1>. Retorna true si se han eliminado correctamente, false en caso contrario.

modificarSimilitud(prod1, prod2, sim) : Modifica la similitud de los productos por la similitud introducida. Primero comprueba que la similitud entre esos dos productos exista y si es así modifica los dos valores que contienen la similitud. Retorna true si se han modificado correctamente, false en caso contrario.

getSimilitud(prod1, prod2) : Consultora de la clase que devuelve la similitud de los productos introducidos. Comprueba su existencia y retorna la similitud en caso afirmativo, al contrario, devuelve -1.0.

getSimilitudes(prods, prod1) : Recorre el vector prods y almacena en un vector de Pair como primer valor el producto que tiene similitud con prod1 y esta como el segundo. Finalmente retorna el vector.

getHmap() : Retorna hmap.

vaciarEstanteria() : Deja hmap vacío.

• Descripción de las relaciones:
- *Almacena:* ControladoraDominio almacena el objeto Similitudes.

3.1.2.5.- Backtracking

Nombre: Backtracking
• Breve descripción de la clase: Clase que se encarga de reordenar los productos en función de la organización de similitudes que se le pasa a su función creadora como parámetro. Usa un algoritmo de Backtracking
• Cardinalidad: *
• Descripción de los atributos:
- *grafo:* Estructura de datos sobre la que se obtiene la solución óptima, se rellena con la traducción del parámetro similitudes de la función creadora.
- *solucion:* Atributo que representa la solución encontrada por el algoritmo.
- *maxSumaPeso*: Atributo usado en el algoritmo. Representa la suma de las aristas del mayor ciclo encontrado hasta el momento.

• Descripción de los métodos:
- *Backtracking*: Función creadora, instancia un backtracking con las similitudes dadas por el parámetro de creación, las traduce al atributo grafo para tener mayor comodidad, ejecuta el backtracking descartando las ramas que no tenga sentido explorar y asigna al atributo solución el valor de retorno del algoritmo.
- *getSol:* Función getter que devuelve el resultado encontrado por el algoritmo

• Descripción de las relaciones:

- Relación “*ejecuta”* con la clase “ControladoraDominio”, indica a qué instancia de la clase controladora del dominio pertenece esta ejecución de Backtracking.

3.1.2.6.- Kruskal

Nombre: Kruskal
• Breve descripción de la clase: Algoritmo que devuelve una solución de la estantería ordenada por la mejor similitud entre productos creando un Árbol de Mínima Expansión. Selecciona las similitudes escogiendo la similitud con el valor más alto y comprueba que añadiendo la similitud al grafo no crea un ciclo. También revisa que el grafo sea un camino en todo momento. No puede haber un producto con más de dos vecinos.
• Cardinalidad: *
• Descripción de los atributos:
- *relationgraph:* Lista de elementos pareja<float<pareja<int,int>> donde el float es el valor de la similitud y los ints son los identificadores de los productos con esa similitud. Se ordena de forma decreciente por valor del float para poder ejecutar Kruskal.

• Descripción de los métodos:
- *Kruskal(Similitudes):* Función creadora de Kruskal. Traduce el HashMap de similitudes y lo convierte en un ArrayList, guardando la información en relationgraph.
- *MST(Cantidad_Productos):* Calcula una solución para ordenar los productos y los devuelve en un ArrayList.
- *getRG():* Devuelve la relationgraph. Utilizada para testing.

• Descripción de las relaciones:
- Relación “*ejecuta”* con la clase “ControladoraDominio”, indica a qué instancia de la clase controladora del dominio pertenece esta ejecución de Kruskal.
- Relación “**usa**” con la clase WQuickUnion indica con qué instancia de WQuickUnion está trabajando Kruskal.

3.1.2.7.- WQuickUnion

Nombre: WQuickUnion
• Breve descripción de la clase: WQuickUnion es una clase que se ocupa de llevar el Merge Find Set. Sirve para comprobar si en el grafo que se está generando, añadiendo los nodos indicados se generaría un ciclo de forma eficiente.
• Cardinalidad: *
• Descripción de los atributos:
- *id:* Vector donde cada posición representa el padre de aquel índice.
- *size:* Vector que contiene el número de nodos en el elemento de ese índice.
- *pointto:* HashMap que guarda la posición de los vectores id y size del producto indicado.

• Descripción de los métodos:
  - *WQuickUnion(size):* Creadora de WQuickUnion. Inicializa los vectores id y size con la medida indicada.
  - *find(i,j);* Retorna un bool indicando si los dos valores tienen el mismo padre. Indica si están conectados o no.
  - *union(i,j):* Inserta los dos elementos a la clase y les da un padre en común.
  - *root(i):* Devuelve el padre del nodo indicado.
• Descripción de las relaciones:
  - Relación "**usa**" con Kruskal. Le indica si los elementos que quiere conectar en el subgrafo que genera ya están conectados.

3.1.2.8.- HillClimbing

Nombre: HillClimbing
• Breve descripción de la clase: Hill Climbing calcula una solución mediante la generación de un estado aleatorio de la ordenación y aplicando operadores para mejorar ese estado. Calcula la calidad del estado mediante una heurística. Cuando no se puede encontrar una mejor solución se devuelve la ordenación con el grado de similitud más alto.
• Cardinalidad: *
• Descripción de los atributos:
  - *graphencypt:* HashMap que relaciona los identificadores de los productos de entrada con un valor entre [0,n). Cada uno con un número único.
  - *graphdecrypt:* HashMap que relaciona un valor entre [0,n) con su identificador de producto correspondiente.
  - *adjMatrix:* Matriz que contiene en la posición [i][j] el grado de similitud de los productos i y j.
  - *count:* Elemento que indica la posición de un producto en la matriz. Empieza igualado a 0 y cada vez que se añade un nuevo producto incrementa por 1.
  - *n:* Cantidad de productos a ordenar.
• Descripción de los métodos:
  - *Heuristic(point):* Calcula el coste de la ordenación presentada.
  - *NeighboringSolutions(currPoint):* Genera otras posibles soluciones mediante una operadora y la ordenación presentada.
  - *Calcrep():* Según la cantidad de productos devuelve la cantidad de vecinos que tendrá la ordenación.
  - *SwapOperator(currPoint.posi,posj):* Dado un vector y dos posiciones dentro del vector devuelve el vector con esos dos elementos intercambiados.
  - *InitialSolutionGenerator():* Genera una ordenación aleatoria de productos para empezar a ejecutar el algoritmo.
  - *translateResult(bstPoint):* Desencripta los valores dentro del vector presentado y devuelve un ArrayList con los productos en la posición correspondiente del vector.

- *HillClimbing(hmap,numproductos):* Creadora de la clase de Hill Climbing. Convierte el hashmap en una matriz donde las filas y las columnas representan la relación y la casilla contiene el valor de la similitud. Al mismo momento guarda la encriptación y desencriptación de los identificadores de los productos con su valor respectivo en los dos hashmaps.
- *ExHillClimbing():* Función que calcula y devuelve una ordenación de la estantería mediante las otras funciones. Calcula la ordenación con la mayor suma de similitudes que encuentra.

• Descripción de las relaciones:
- Relación "*ejecuta*" con la clase "ControladoraDominio", indica a qué instancia de la clase controladora del dominio pertenece esta ejecución de Hill Climbing.

3.1.2.9.- Usuario

Nombre: Usuario
• Breve descripción de la clase: Clase que simula un usuario
• Cardinalidad: * (un usuario para cada controladora del dominio)
• Descripción de los atributos:
- *username:* String que representa el nombre de usuario.
- *password:* String que representa la contraseña del usuario.

• Descripción de los métodos:
- *Usuario(user, pwd)*: Función creadora, instancia un objeto de la clase usuario con el nombre y contraseña que se pasan como parámetros.
- *entraContrasena(pwd)*: Función que dada una contraseña devuelve un booleano indicando si coincide con la contraseña actual de ese usuario.
- *getUsername()*: Función getter que devuelve el nombre de usuario del Usuario.
- *getPassword()*: Función getter que devuelve la contraseña del Usuario.

• Descripción de las relaciones:
- Relación "*contiene*" con la clase "CjtUsuarios": Indica qué usuarios hay registrados en el sistema.
- Relación *"usa"* con la clase "ControladoraDominio", indica la instancia de la controladora que tiene asignado ese usuario.

3.1.2.10.- Controladora Dominio

Nombre: ControladoraDominio (En la primera entrega era CjtUsuarios)
• Breve descripción de la clase: Clase que se encarga de la gestión de usuarios. Por cada usuario tiene una controladora de usuario, que se encarga de gestionar la estantería de ese usuario y las similitudes de sus productos
• Cardinalidad: 1
• Descripción de los atributos:

- *userController():* Estructura de datos que almacena las controladoras disponibles con el nombre de usuario como key.
- *usuarioActivo:* Usuario activo en ese momento, null si no se ha iniciado sesión.
- *controladoraActiva:* Controladora del dominio activa en ese momento, null si no se ha iniciado sesión.

• Descripción de los métodos:
- *CjtUsuarios*: Función creadora.
- *SignUp*: Da de alta el usuario que se pasa como parámetro si este no existe.
- *login*: Valida el usuario y contraseña que se dan como parámetro, si coinciden se inicia sesión.
- *logOut*: Cierra sesión del usuario que haya activo, muestra un mensaje de error si no hay ningún usuario activo.
- *printUsuarioActual*: Muestra por la entrada/salida estándar el usuario con la sesión iniciada actualmente, en caso que no haya ninguna sesión iniciada, también lo muestra.
- *loggedIn*: Devuelve un booleano indicando si hay un usuario activo en ese momento.
- Los métodos que siguen son simple paso de parámetros del driver a la función controladora del dominio, por lo que no los nombramos aquí, ya se hará a continuación en Controladora Dominio
  Todos estos métodos son los usados para ejecutar las funcionalidades del proyecto.

• Descripción de las relaciones:
- Relación "*contiene*" con la clase "Usuarios": Indica qué usuarios hay registrados en el sistema.

3.1.2.11.- Controladora Usuario

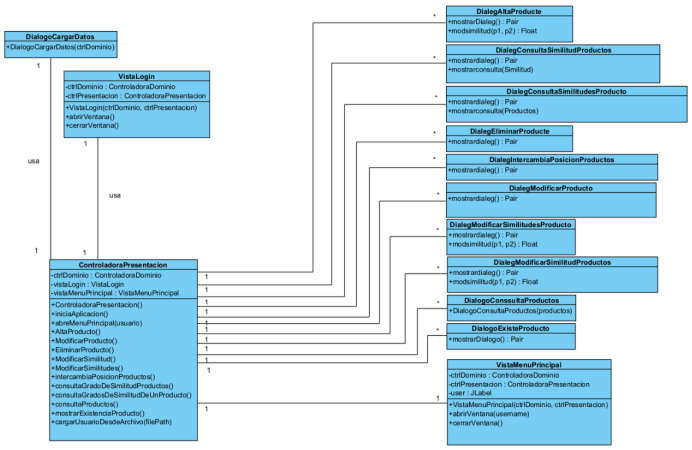
Nombre: ControladoraUsuario (En la primera entrega era ControladoraDominio)
• Breve descripción de la clase: .Clase contenedora de los objetos Estantería y Similitudes. Permite gestionar a la vez esos dos objetos, permitiendo así el acceso tanto a los productos, estantería y las similitudes entre los primeros.
• Cardinalidad: * Una por cada Usuario
• Descripción de los atributos:
- Username: Nombre de usuario del usuario que controla.
- Password: Contraseña del usuario que controla.
- Estantería: objeto de la clase Estantería. Esta simula una estantería donde se almacenan productos
- Similitudes: objeto de la clase Similitudes. Esta almacena las similitudes de los productos que se encuentran en la estantería.

• Descripción de los métodos:
- *ControladoraDominio()* : Creadora de la clase. Asigna a los atributos sus respectivos objetos no inicializados.

- *anadirProductoEstanteria(p)* : Comprueba si el producto ya existe, si no lo hace lo añade a la estantería.
- *anadirSimilitudProducto(prod1, prod2, sim)* : Añade al objeto Similitudes la similitud de ambos productos.
- *eliminarProductoEstanteria(p)* : Comprueba si el producto existe, si es así lo elimina de la estantería junto todas sus similitudes.
- *ordenarEstanteria(num)* : Ordena la estantería según el número introducido. 1 para Backtracking y 2 para HillClimbing. También comprueba errores.
- *modificarSimilitudProductos(p1, p2, sim)* : Modifica las similitudes de ambos productos.
- *getSimilitudProductos(p1, p2)* : Comprueba que ambos productos existen en la estantería y retorna su similitud.
- *getSimilitudesProducto(p)* : Printa en pantalla todas las similitudes de los productos con el producto p.
- *intercambiarProductos(p1, p2)* : Intercambia la posición de p1 y p2.
- *validarProductos(p1, p2)* : Valida si los dos productos introducidos existen en la estantería, retornando un Pair de Booleans, dónde first y second representan la validez de cada producto respectivamente.
- *existeProducto()* : Comprueba si un producto existe, si lo hace retorna sí ID, si no -1.
- *vaciarEstanteria()* : Vacía la estantería eliminando todos sus productos de Estantería y sus similitudes de Similitudes.

3.2 Capa de presentación

3.2.1- Diagrama de clases de diseño de la Presentación



3.2.2-Breve descripción de las clases

Todos los diálogos no descritos aquí ejecutan los casos de uso explicados anteriormente.

3.2.2.1 Dialogo Cargar Datos
Esta clase abre un diálogo que permite al usuario cargar todos los datos de ejecuciones anteriores o no, si quiere empezar de cero.

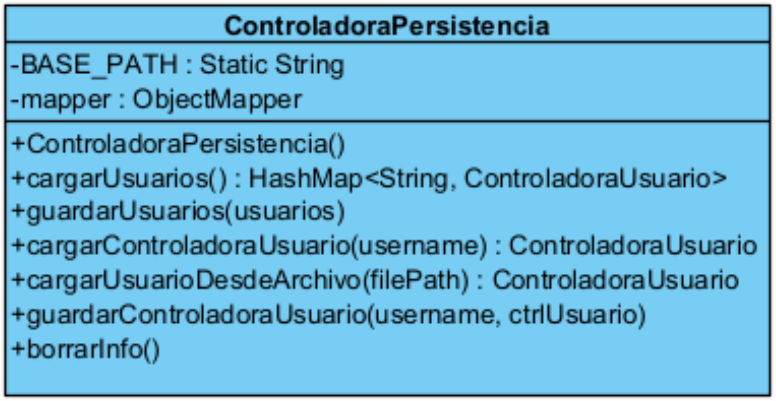
3.2.2.2 Vista Login
Esta clase es una extensión de frame y se ocupa de acceder a los datos de un usuario mediante el “login” o crear un nuevo usuario al hacer “sign up”. También permite cargar datos de un fichero .json y eliminar la base de datos.

3.2.2.3 Vista Menu Principal
Esta clase es una extensión de frame y presenta el menú principal del sistema con todas sus funcionalidades para la gestión de la estantería, los productos y sus similitudes.

3.2.2.4 Controladora presentación
Esta clase se ocupa del funcionamiento correcto de la capa de presentación. Es la que se ocupa de abrir los diálogos pertinentes de la función seleccionada en el menú por el usuario, obtener los datos e insertarlos en el sistema.

3.3.- Capa de persistencia

3.3.1.- Diagrama de clases de diseño de la capa de persistencia



3.3.2.- Breve descripción de la clase

Nombre: ControladoraPersistencia
• Breve descripción de la clase: Se encarga de almacenar y recuperar información. La información se almacena en un fichero para cada usuario.
• Cardinalidad: 1
• Descripción de los atributos:
- *BASE_PATH:* String que representa la ubicación del directorio donde se encuentran los ficheros “usuario.json”, tiene el valor *"src/main/usuarios/"*.
- *mapper:* Objeto que sirve para leer y escribir en cada fichero .json.

• Descripción de los métodos:
- *ControladoraPersistencia()*: Función creadora.
- *cargarUsuarios()*: Función que lee todos los usuarios en la ubicación indicada por BASE_PATH.
- *guardarUsuarios()*: Función que se encarga de escribir en ficheros .json ubicado en BASE_PATH los usuarios que tenemos en el sistema.
- *cargarControladoraUsuario(username)*: Función que lee un usuario desde su archivo en BASE_PATH.
- *cargarUsuarioDesdeArchivo(filePath):* Función que lee un usuario desde el archivo dado como parámetro y que en caso que este tenga un formato correcto, se añade al sistema y al directorio BASE_PATH
- *guardarControladoraUsuario(username,ctrlUsuario):* Función que se encarga de escribir en un fichero .json en BASE_PATH el la información del usuario pasada como parámetro
- *borrarInfo():* Función que se encarga de borrar toda la información almacenada en BASE_PATH. Se usa para empezar desde cero todo el sistema.

• Descripción de las relaciones:
- Relación “*es mantenida por*” con la clase “ControladoraDominio”: Indica controladora del dominio está manteniendo esta controladora de persistencia.

4.- Estructuras de datos usadas

En la mayoría de clases hemos usado las siguientes estructuras:

- **Pair<K, V>**

  Al querer almacenar los productos en parejas para facilitar así la gestión de sus similitudes, tuvimos que crear una clase Pair que consta de una estructura de datos con dos elementos, first y second y sus respectivas operaciones constructoras, modificadoras, consultoras y de escritura.

- **ArrayList<T>**

Usamos el arraylist como una lista o un vector como estructura de datos dinámica, nos sirve para poder representar los productos por ejemplo, ya que nos interesa poder añadir y/o eliminar productos cómodamente. En ese caso tenemos un ArrayList<Integer>, ya que nos permite añadir y eliminar los enteros fácilmente.

- **HashMap<K,V>**

  También hemos usado el hashmap como diccionario en reiteradas ocasiones. Esta estructura de datos nos sirve para acceder a elementos de una forma sencilla y eficaz. Nos sirve para la estantería, similitudes, backtracking e incluso para el algoritmo de Kruskal, ya que nos ofrece un acceso eficiente a elementos indexados por una Key.

- **Vector<T>**

  Hemos usado un vector en Similitudes como estructura de datos para devolver todas las similitudes de un solo producto, para poder ser leídas por el Main.

Hay clases como las de los algoritmos, por ejemplo, en las que hemos usado conjuntos como estructuras de datos para no tener elementos duplicados.

- **Set<T>**

  Usamos sets cuando queremos tener una estructura de datos que nos mantenga un conjunto de elementos que no mantiene duplicados. Este conjunto también nos garantiza que estos elementos estarán ordenados. Lo hemos usado en Backtracking, por ejemplo.

- **HashSet<T>**

  Usamos hash sets, en cambio, cuando no nos importe el orden en el que se almacenan los elementos. Lo hemos usado también en Backtracking.

En ambos casos usamos conjuntos para evitar duplicados, uno te garantiza el orden y el otro no, elegir uno de ellos ha dependido de nuestras necesidades en ese momento.

5.- Algoritmos Usados

5.1.- Backtracking

Para programar el Backtracking hemos aprovechado la geometría de la estantería de tamaño n para, a partir del primer elemento, permutar los n-1 elementos restantes. Al final, devolver [1,2,3] es equivalente a devolver [3,1,2] o [2,3,1], ya que al ser una estantería circular, el primer y el último elemento estarán conectados. Con esta idea en mente, el algoritmo puede coger el primer elemento e iniciar el backtracking desde allí.

En este backtracking se prueban todas las combinaciones posibles partiendo del primer nodo mientras se van descartando las combinaciones que no pueden ser candidatas a solución.

La ejecución de este algoritmo es fácil de ver al considerar todas las permutaciones y ver cómo el algoritmo las recorre. Vamos a ver cómo el algoritmo puede descartar ciertas ramas.

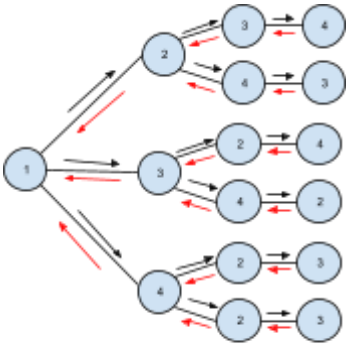


Figura 1

En la *figura 1* podemos ver las permutaciones posibles de una estantería con 4 productos con IDs [1,2,3,4]. En este caso, el algoritmo recorre en primera instancia la rama de nodos 1,2,3,4 sumando los pesos de las aristas recorridas. Al llegar al nodo 4 se obtiene la primera solución, (en la que se tiene en cuenta la arista 4-1). Al haber llegado a una primera solución el algoritmo tiene en cuenta su peso. Podremos descartar caminos que aún no hemos terminado de recorrer cuando veamos que el ciclo que estamos recorriendo tiene siempre peso menor a la suma máxima encontrada hasta el momento. Esta idea, llamada pruning, la podemos usar porque sabemos que siempre tendremos aristas con peso en $[0.0, 1.0]$. Por ejemplo, cuando nos quedan 2 nodos por visitar nos quedan, también, 3 aristas. Si tenemos peso actual 0.2, el valor máximo al que se puede llegar en esta solución parcial es 3.2, (pues en mejor caso tendremos pesos (1.0+1.0+1.0 = 3), si en ese momento ya hemos encontrado una solución con valor 4.0, no nos interesa seguir recorriendo esa rama del grafo, por lo que podemos “tirar para atrás”, siguiendo las flechas rojas en la *figura 1*.

Estructuras de datos usadas en Backtracking:

1) **Atributo grafo**

   Hemos usado un *“HashMap<Integer, ArrayList<Pair<Integer, Float>>>”* Para representar este grafo de similitudes entre productos, donde los los ids son nodos y las relaciones entre productos aristas con su correspondiente peso.

   El HashMap ha sido nuestra estructura de datos porque nos ha parecido un formato eficiente para una lista de adyacencias, ya que nos permite acceder a la lista de aristas de un nodo determinado fácilmente. La lista de nodos ha sido un ArrayList porque es una estructura de datos que hemos usado mucho y nos pareció la opción más cómoda.

2) **Atributo solución**

   Nuestra solución (el ciclo con mayor peso) se almacena en una ArrayList<Integer> simplemente porque así lo tiene como atributo la estantería.

3) **Atributo maxSumaPeso**

   Nuestro valor máximo lo hemos almacenado en un Float porque, al ser un valor decimal entre 0.0 y 1.0, sólo podíamos escoger un Float o un Double. El Double ocupa más espacio, por lo que el Float ha sido la opción escogida.

4) **Variables locales**

   Para mantener constancia de los elementos visitados, hemos usado un HashSet, ya que nos mantiene elementos con acceso eficiente por valor y no nos los ordena. En este caso en concreto, no nos interesaba ordenar los conjuntos de nodos.

5.2.- Kruskal

El algoritmo de Kruskal parte de una simple idea. Teniendo un grafo no dirigido, conexo, donde cada arista tiene un peso, se puede conseguir el subgrafo árbol de mínima expansión con el coste mínimo construyendo el subgrafo desde cero, insertando la arista con el peso más pequeño de uno en uno hasta llegar a un MST (minimum spanning tree) con las aristas de mínimo peso. Para asegurar que se genere un MST el algoritmo comprueba

que la arista que se añadirá no genere un ciclo. Si ese es el caso, la arista se descarta.

Como nosotros queremos que el árbol esté formado por las aristas de mayor peso en vez de ordenar las aristas de forma creciente las ordenamos en decreciente. De esta forma siempre seleccionamos las de mayor peso.

A la hora de evitar la generación de ciclos se utiliza un Merge Find Set con una clase llamada WQuickUnion. Nos permite comprobar de forma eficiente si al añadir una arista nos genera un ciclo.

Nuestro objetivo de obtener una estantería ordenada causa que tengamos un nivel de dificultad añadido. Los productos deben estar ordenados formando un camino cíclico, donde cada producto tiene un anterior y un posterior. Kruskal no genera un path sinó que genera un árbol. Se puede condicionar el algoritmo a que crea un camino entre productos. Por eso el algoritmo que utilizamos chequea la cantidad de veces que se añade una arista en el subgrafo que se está generando comprobando que no aparezca más de dos veces, asegurando la condición de que se genere un camino como solución. Esta comprobación sólo se ejecutará siempre que una arista sea aceptada por las condiciones del algoritmo de Kruskal.

Explicado de otra forma, solo se puede insertar un nodo al grafo si solo si se està conectando a un nodo exterior. A un nodo que no esté conectado ya a dos otros nodos.

Estructuras de datos en Kruskal

1) ArrayList: Se utiliza un ArrayList en dos casos para la misma función: Tener en secuencia a los productos. Relationgraph tiene parejas de productos ordenados decreciente por sus similitudes para la selección de aristas y result contiene los identificadores de los productos en el orden encontrado por el algoritmo.
2) HashMap: El HashMap se ha utilizado para poder guardar las similitudes de los productos que ya se habían dado por válidas y por que es más eficiente al buscar qué elementos contiene.
3) WQuickunion: Aunque sea una clase en verdad es una estructura de datos que es eficiente en encontrar si dos nodos ya están conectados y por eso se utiliza.

5.3.- HillClimbing

El algoritmo de Hill Climbing contiene varias partes:
La generadora de la solución inicial.

La operadora que genera los vecinos de la solución sobre la cual se opera.
La Heurística que calcula la corrección de la solución presentada.
Primero se genera una solución inicial. En nuestro caso se crea un orden aleatorio de productos en la estantería. Sobre la ordenación aleatoria se generan sus vecinos. Los vecinos se generan aplicando sobre la solución la operadora. Nosotros utilizamos una operadora que ejecuta un swap. Intercambia dos productos dentro de la solución. Este swap se ejecuta con todas las posibles combinaciones dentro de la solución generando todos los vecinos posibles. Ahora, con la solución y sus vecinos se calcula todos sus costes, siendo la suma de las aristas que aparecen en la ordenación. Se queda con la mejor ordenación. Si la mejor ordenación era un de los vecinos se vuelve a iterar, esta vez la solución siendo el mejor vecino. Si la mejor solución sigue siendo la solución anterior se ha encontrado un máximo local y se termina la búsqueda, retornando la última solución generada.

Estructuras de datos en Hill Climbing

1) ArrayList: Se utilizan los ArrayList en Hill Climbing para guardar los productos en un orden generado. Cuando solo se quiere tener una lista de productos.
2) HashMap: El HashMap se utiliza para guardar una relación entre el identificador del producto y un valor que se le asigna al crear la instancia de Hill Climbing. Para evitar errores nos permite asignarle un identificador especial al identificador del producto que sea menor que el número de productos para que se pueda insertar en una matriz sin dar errores. Algo que los identificadores de los productos no nos pueden asegurar.
3) LinkedList: Los LinkedList son utilizados para guardar todas las instancias de las combinaciones de los productos. Una lista de listas. Preferidas antes de un ArrayList al guardar varias listas.