Programación Orientada a Objetos

Sara Cubero García-Conde

Índice:

1. **Análisis de la aplicación realizada.**
    a. **Estrategias implementadas y Decisiones de diseño establecidas.**
2. **Diagrama de clases.**
3. **Descripción de la aplicación.**
4. **Anexo con el código fuente de las clases implementadas.**

1. Análisis de la aplicación realizada.

La aplicación tiene como objetivo facilitar la gestión de una cooperativa agrícola, uniendo a pequeños productores rurales y ayudándoles a distribuir y vender sus productos. La cooperativa se compone de varios actores, como productores, logística, distribuidores y consumidores finales.

Los productores se dividen en pequeños y grandes productores, así como productores federados, que unen sus productos específicos para ser considerados como un solo productor. Cada producto tiene un rendimiento por hectárea y un valor de referencia por kilogramo libre de impuestos.

La logística se encarga de transportar los productos a los clientes finales y distribuidores, dividiéndose en productos perecederos y no perecederos. Los costos de logística dependen del tipo de producto y la distancia de transporte, así como de la gran y pequeña logística necesaria para llegar al destino.

Los distribuidores y consumidores finales compran productos de la cooperativa a precios que incluyen un margen y los costos de logística. Los distribuidores compran grandes cantidades, mientras que los consumidores finales compran directamente de la cooperativa en cantidades más pequeñas.

La aplicación debe ser desarrollada utilizando el paradigma de Programación Orientada a Objetos en Java con BlueJ y modelar la gestión de la cooperativa de manera genérica y flexible.

Estrategias implementadas y decisiones de diseño establecidas.

Para llevar a cabo la aplicación, se han utilizado diferentes estrategias y se han tomado decisiones de diseño para garantizar la eficiencia, mantenibilidad y escalabilidad del software. Las estrategias implementadas y las decisiones de diseño establecidas han contribuido a un código más estructurado, legible, reutilizable y escalable. Esto facilita el mantenimiento y la evolución de la aplicación a largo plazo, permitiendo la incorporación de nuevas funcionalidades y adaptándose a posibles cambios en los requisitos del sistema.

En términos de estrategias implementadas, se ha hecho uso de patrones de diseño para mejorar la estructura del código y la calidad del software. Entre los patrones utilizados se encuentran el patrón Singleton, que ha sido empleado en la clase de límite de hectáreas para garantizar la existencia de una única instancia de la clase en todo el sistema, y el patrón de diseño Factory, utilizado en la creación de objetos de logística para simplificar y centralizar el proceso de creación.

Otra estrategia empleada ha sido la programación defensiva, que consiste en anticiparse a posibles errores o situaciones imprevistas. Esto se ha logrado mediante el uso de mecanismos como la validación de entradas de usuario para asegurar que los datos ingresados sean correctos y coherentes, y el manejo de excepciones para detectar y gestionar errores de manera adecuada, evitando la interrupción abrupta del programa y brindando información útil sobre el error.

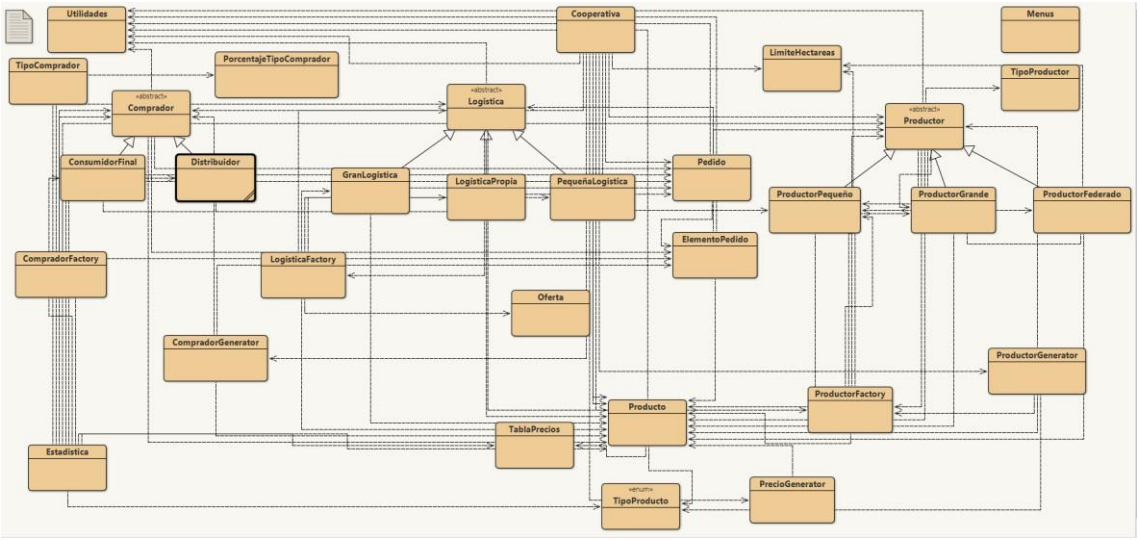
Asimismo, se ha dado importancia a la legibilidad y comprensión del código a través de comentarios y documentación. Los comentarios se han utilizado para explicar la lógica y funcionamiento de las diferentes secciones del código, facilitando su entendimiento y futuras modificaciones. La documentación también ha sido útil para describir el propósito de las clases

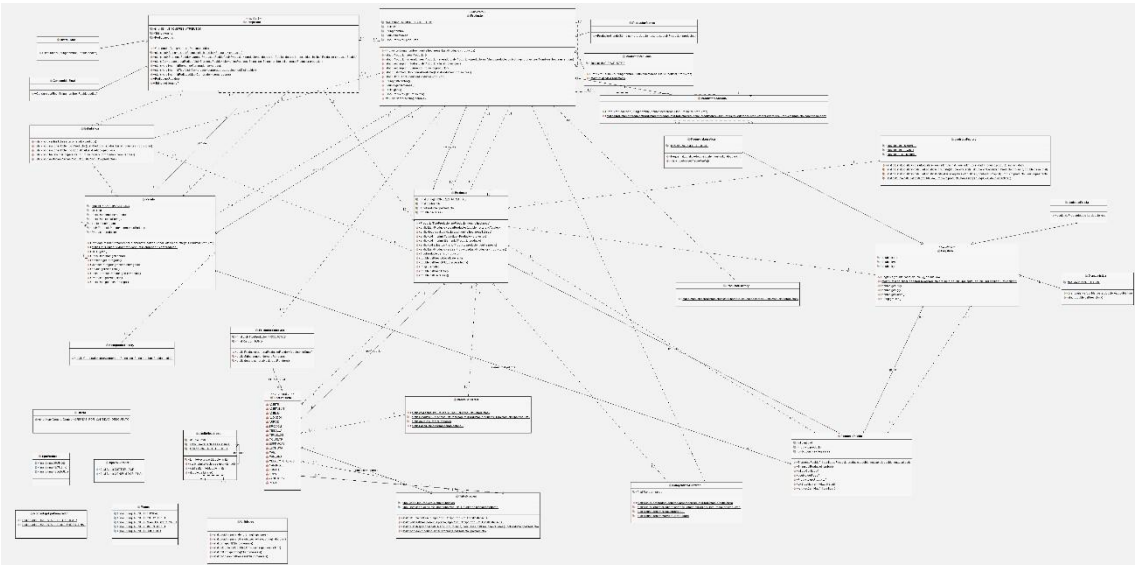
y métodos, así como las interfaces y parámetros utilizados, promoviendo un mejor entendimiento del software en su conjunto.

Otra estrategia importante ha sido la reutilización de código. Se han implementado métodos estáticos reutilizables que encapsulan lógica común y evitan la duplicación de código. Esto ha contribuido a la eficiencia del desarrollo y a mantener un código más limpio y organizado.

Por último, se han utilizado clases para la generación de elementos random, lo que permite generar datos de manera aleatoria de forma controlada y consistente, garantizando la variedad y realismo en los escenarios de prueba.

2. Diagrama de clases.





No.	Clase	Descripción
1	Comprador.java	Clase abstracta que tiene dos subclases: ConsumidorFinal y Distribuidor.

2	CompradorFactory.java	Clase que se encarga de crear instancias de la clase Comprador.

3	CompradorGenerator.java	Clase que se encarga de generar nombres aleatorios para los compradores.
4	ConsumidorFinal.java	Hereda de la clase Comprador y proporciona un constructor para crear un objeto ConsumidorFinal.
5	Cooperativa.java	Clase que representa una cooperativa con productores y compradores.
6	Distribuidor.java	Hereda de la clase Comprador y proporciona un constructor para crear un objeto Distribuidor.
7	ElementoPedido.java	Clase que representa un elemento en un pedido, tiene una relación de composición con la clase Producto y Logistica.
8	Estadistica.java	Interfaz que define métodos para calcular estadísticas sobre los pedidos realizados por los compradores.
9	GranLogistica.java	Subclase de la clase Logistica, representa el envío de productos a largas distancias.
10	LimiteHectareas.java	Interfaz que define métodos para verificar si un productor ha superado el límite máximo permitido de hectáreas cultivadas.
11	Logistica.java	Clase abstracta que tiene dos subclases: GranLogistica y PequeñaLogistica, representa el envío de productos a los compradores.
12	LogisticaFactory.java	Clase que se encarga de crear instancias de la clase Logistica.
13	LogisticaPropia.java	Hereda de la clase Logistica, representa el envío de productos por parte del productor.
14	Menus.java	Clase que define métodos para mostrar menús en la consola.

15	Oferta.java	Clase que representa una oferta de un productor para vender sus productos a un precio determinado.

16	Pedido.java	Clase que representa un pedido realizado por un comprador, tiene una relación de composición con la clase ElementoPedido.
17	PequeñaLogistica.java	Subclase de la clase Logistica, representa el envío de productos a cortas distancias.
18	PorcentajeTipoComprador.java	Interfaz que define métodos para calcular el porcentaje de un tipo de comprador en una lista de compradores.
19	PrecioGenerator.java	Clase que se encarga de generar precios aleatorios para los productos.
20	Producto.java	Clase que representa un producto cultivado por un productor, tiene una relación de composición con la clase TipoProducto.
21	Productor.java	Clase abstracta que tiene tres subclases: ProductorFederado, ProductorGrande y ProductorPequeño.
22	ProductorFactory.java	Clase que se encarga de crear instancias de la clase Productor.
23	ProductorFederado.java	Hereda de la clase Productor y representa a un productor federado.
24	ProductorGenerator.java	Clase que se encarga de generar nombres aleatorios para los productores.
25	ProductorGrande.java	Hereda de la clase Productor y representa a un productor grande.
26	ProductorPequeño.java	Hereda de la clase Productor y representa a un productor pequeño.
27	Resultado.java	Interfaz que define métodos para mostrar resultados en la consola.

28	TablaPrecios.java	Clase que representa una tabla con los precios de los productos ofrecidos por los productores, tiene relaciones de composición con las clases Producto y Oferta.
29	TipoComprador.java	Enumeración que define los tipos posibles de compradores: CONSUMIDOR_FINAL y DISTRIBUIDOR.
30	TipoProducto.java	Enumeración que define los tipos posibles de productos.
31	TipoProductor.java	Enumeración que define los tipos posibles de productores: FEDERADO, GRANDE y PEQUEÑO.
32	Utilidades.java	Clase que define métodos útiles para el programa.

3. Descripción de la aplicación.

Comprador: representa un comprador, tiene los siguientes atributos:

nombre: es el nombre del comprador. pedido: es

el pedido realizado por el comprador.

Métodos:

extraerProductosUnicos(productos): Esta función toma una lista productos de productos y devuelve una lista de nombres únicos de los productos.

agregarProductosPedido(productores, kmCapital, kmLocalidad, tipo): Esta función permite al usuario seleccionar un productor, un producto y una cantidad. Toma una lista productores de productores disponibles, kmCapital que es la distancia en kilómetros a la capital, kmLocalidad que es la distancia en kilómetros a la localidad, y tipo que es el tipo de comprador (1: Consumidor Final, 2: Distribuidor). Devuelve una lista de elementos de pedido creados.

crearPedido(elementosPedidos, tipo, nombre, entrega, productor): Esta función toma una lista elementosPedidos de elementos de pedido, tipo que es el tipo de comprador (Consumidor Final o Distribuidor), nombre que es el nombre del comprador, entrega que son los días de entrega del pedido y productor. Devuelve un objeto Comprador que representa el pedido creado.

imprimirElemento(comprador): Esta función imprime los atributos de un objeto comprador.

imprimirTipo(compradores, esDistribuidor): Esta función imprime los compradores de un tipo específico. Toma una lista compradores de compradores y un booleano esDistribuidor que indica si se deben imprimir los compradores de tipo Distribuidor.

imprimirPedidos(compradores): Esta función imprime los pedidos disponibles y muestra información detallada sobre un pedido seleccionado. Toma una lista compradores de compradores.

getPedido(): Este método devuelve el objeto Pedido asociado al comprador. getNombre():

Este método devuelve el nombre del comprador. crearComprador(tipo, nombre, pedido):

Esta fábrica crea un objeto Comprador según el tipo especificado (DISTRIBUIDOR o

CONSUMIDOR_FINAL), nombre del comprador y pedido asociado al comprador.

generarCompradores(productores): Esta fábrica genera una lista de compradores aleatorios basados en la lista productores de productores disponibles.

agregarElementosPedido(productos): Este método agrega elementos de pedido aleatorios a una lista productos de productos disponibles y devuelve una lista de elementos de pedido generados aleatoriamente.

generarTipoComprador(): Esta función genera aleatoriamente un tipo de comprador, que puede ser "DISTRIBUIDOR" o "CONSUMIDOR_FINAL". Devuelve el tipo de comprador generado.

generarNombreComprador(): Esta función genera aleatoriamente un nombre de comprador utilizando una combinación de nombres y apellidos. Devuelve el nombre de comprador generado.

ConsumidorFinal: Esta es una subclase de la clase "Comprador" que representa a un comprador de tipo "Consumidor Final". Tiene un constructor ConsumidorFinal(nombre, pedido) que toma el nombre del comprador y el pedido del comprador para crear un nuevo objeto de tipo "ConsumidorFinal".

Cooperativa: Esta clase representa una cooperativa que contiene productores y compradores.

Contiene dos listas: productores y compradores para almacenar los productores y

compradores respectivamente. Esta clase contiene varios métodos, incluyendo: main(args):

Este es el punto de entrada principal de la aplicación. menuPrincipal(): Este método muestra

el menú principal y maneja la entrada del usuario. menuProductor(): Este método presenta un

menú para interactuar con objetos "Productor".

menuCrearProductor(): Este método presenta un menú para crear nuevos objetos "Productor".

crearProductorFederado(): Este método crea un nuevo "Productor Federado" a partir de varios "Productores Pequeños".

menuLimiteHectareas(): Este método presenta un menú para establecer un límite en las hectáreas.

menuCompradores(): Este método presenta un menú para interactuar con objetos "Comprador".

menuEstadistica(): Este método presenta un menú para interactuar con las estadísticas.

cargaPrincipal(): Este método carga los datos iniciales de la aplicación, crea productores y compradores de prueba y actualiza los precios de los productos.

verificarEntregasYPrecios(fechaAutomatica, imprimir): Este método verifica las entregas y los precios en una fecha determinada. Si se proporciona una fecha (fechaAutomatica), verifica las

entregas y precios en esa fecha. Si no se proporciona una fecha, verifica las entregas y precios en la fecha actual. El parámetro "imprimir" indica si los resultados deben imprimirse o no.

Distribuidor: Esta es una subclase de la clase "Comprador" que representa a un comprador de tipo "Distribuidor". Tiene un constructor Distribuidor(nombre, pedido) que toma el nombre del comprador y el pedido del comprador para crear un nuevo objeto de tipo "Distribuidor".

ElementoPedido: Esta clase representa un elemento de un pedido. Cada objeto "ElementoPedido" tiene una cantidad, un producto y una lista de objetos "Logística" asociados a él. La clase tiene varios métodos, incluyendo:

ElementoPedido(cantidad, producto, kmCapital, kmLocalidad): Es el constructor de la clase.

Crea un nuevo elemento de pedido con la cantidad y el producto especificados, y los

kilómetros a la capital y a la localidad. getProductoNombre(): Devuelve el nombre del

producto del elemento del pedido. getCantidad(): Devuelve la cantidad del producto en el

elemento del pedido. getPeso(): Devuelve el peso total del elemento del pedido.

getProducto(): Devuelve el producto del elemento del pedido.

getLogistica(): Devuelve la lista de logísticas asociadas al elemento del pedido.

setCantidad(cantidad): Establece la cantidad del producto en el elemento del pedido.

Estadisticas: Esta clase contiene varios métodos para calcular diversas estadísticas relacionadas con los pedidos, ventas y precios:

ventasTotales(pedidos): Este método calcula las ventas totales por producto en un período de tiempo específico. Toma una lista de pedidos como entrada.

importesObtenidosProductor(pedidos, productores): Este método calcula los ingresos obtenidos por cada productor en función de los pedidos entregados. Toma una lista de pedidos y una lista de productores como entrada.

importesObtenidosLogistica(pedidos): Este método calcula los importes obtenidos por cada tipo de logística en función de los pedidos entregados. Toma una lista de pedidos como entrada.

beneficiosCooperativaPorProducto(compradores): Este método calcula los beneficios totales obtenidos por la cooperativa por cada producto. Toma una lista de compradores como entrada.

evolucionPreciosProducto(productos): Este método calcula la evolución de los precios de cada producto. Toma una lista de productos como entrada.

GranLogistica: Esta es una subclase de la clase "Logistica" que representa una logística de gran tamaño para el transporte de productos. Tiene un constructor GranLogistica(coste, kg, km) que toma el coste, el peso en kg y la distancia en km de la logística para crear un nuevo objeto de tipo "GranLogistica". También tiene un método getPrecioKm() que obtiene el precio por kilómetro de la logística de gran tamaño.

LimiteHectareas: Esta clase representa el límite máximo de hectáreas que un productor puede tener en la cooperativa empresarial. Esta clase es implementada como un patrón Singleton para garantizar que solo exista una instancia del límite de hectáreas. Tiene varios métodos, incluyendo:

getInstance(): Método para obtener la instancia única de la clase.

setLimite(limite): Método para establecer un nuevo límite. getLimite():

Método para obtener el límite actual.

Logistica: Esta es una clase abstracta que representa una entidad de logística. Tiene un constructor Logistica(coste, kg, km) que toma el coste, los kilogramos de carga y los kilómetros recorridos en la logística para crear un nuevo objeto de tipo "Logistica". Además, tiene varios métodos, incluyendo:

generarEnvio(producto, kmCapital, kmLocalidad, cantidad): Método estático para generar un envío de logística. Crea un envío utilizando la fábrica de logística. getKm(): Obtiene los kilómetros recorridos en la logística. getKg(): Obtiene los kilogramos de carga de la logística. getCoste(): Este método obtiene el coste de la logística. getTipo(): Este método obtiene el tipo de logística, devolviéndolo como una cadena de texto.

LogisticaFactory: Esta es una clase que se encarga de crear objetos de logística según los parámetros proporcionados. Tiene varios métodos, incluyendo:

crearLogistica(kmCapital, kmLocalidad, producto, cantidad): Crea una lista de objetos de logística según los parámetros especificados.

calcularLogisticaConCoste(kmCapital, kmLocalidad, producto, pesoTotal): Calcula la logística con coste y devuelve una lista de objetos de logística.

calcularLogisticaPorKg(logisticas, kmCapital, producto, pesoTotal): Calcula la logística por kilogramo y agrega los objetos de logística a la lista existente.

calculaCoste(km, producto, granLogistica, cantidad): Calcula el costo de una operación de logística en función de los kilómetros, el producto, si es una operación de logística de gran tamaño y la cantidad.

LogisticaPropia: Esta es una subclase de la clase "Logistica" que representa una instancia de logística propia. Tiene un constructor LogisticaPropia(kg, km) que toma el peso en kilogramos y el recorrido en kilómetros para crear un nuevo objeto de tipo "LogisticaPropia".

OFERTAS_POR_CANTIDAD_DESCUENTO: Este es un mapa que almacena las ofertas por cantidad de productos y su respectivo descuento. La clave representa la cantidad de productos y el valor representa el descuento. Se agregan al mapa tres ofertas con sus respectivos descuentos.

Menus: Esta es una clase que contiene arrays de strings que representan distintos menús para el usuario. Contiene varios arrays de strings, incluyendo MENU_PRINCIPAL, MENU_PRODUCTORES, MENU_CREAR_PRODUCTORES, MENU_COMPRADORES y MENU_ESTADISTICA que representan respectivamente el menú principal de la aplicación, el menú de gestión de productores, el menú de creación de productores, el menú de gestión de distribuidores y consumidores finales y menú de gestión de estadística.

Pedido: Esta es una clase que representa un pedido realizado por un productor. Contiene varios campos para almacenar información sobre el pedido y varios métodos, incluyendo:

Pedido(elementosPedidos, fechaEntrega, productor): Este es el constructor de la clase. Crea un nuevo pedido con los elementos del pedido, la fecha de entrega y el productor que realizó el pedido.

extraerPedidos(compradores): Este método extrae todos los pedidos de todos los compradores y los devuelve en una lista. getId(): Este método obtiene el identificador único del pedido. getFecha(): Este método obtiene la fecha y hora en que se realizó el pedido.

getEntregado(): Este método indica si el pedido ha sido entregado, devolviendo true si ha sido entregado y false en caso contrario. setEntregado(entregado): Este método establece el estado de entrega del pedido. getPesoTotal(): Este método obtiene el peso total del pedido.

getElementos(): Este método obtiene la lista de elementos de pedido. getProductor(): Este método obtiene el productor que realizó el pedido. getFechaEntrega(): Este método obtiene la fecha de entrega del pedido.

PequeñaLogistica: Esta es una subclase de la clase "Logistica" que representa la logística de tipo pequeña. Tiene un constructor PequeñaLogistica(coste, kg, km) que toma el costo de la logística, el peso en kilogramos y los kilómetros recorridos para crear un nuevo objeto de tipo "PequeñaLogistica".

getPrecioKmKg(): Este método obtiene el precio por kilómetro por kilogramo para la logística pequeña.

PorcentajeTipoComprador: Esta es una clase que define los porcentajes para cada tipo de comprador. Proporciona constantes estáticas para el porcentaje de distribuidor y el porcentaje de consumidor final.

PrecioGenerator: Esta es una clase que se encarga de generar y actualizar precios únicos para los productos. Contiene varios métodos, incluyendo:

actualizarPreciosUnicos(productos): Este método actualiza los precios únicos de los productos en la tabla de precios.

buscarProductoPorTipo(productos, tipoProducto): Este método busca un producto en la lista de productos por su tipo y devuelve el producto encontrado o null si no se encuentra.

numeroRandom(): Este método genera un número aleatorio entre 0.1 y 1 con dos decimales.

generarFechaAleatoria(): Este método genera una fecha aleatoria dentro del año actual.

Producto: Esta es una clase que representa un producto y sus atributos. Tiene varios campos para almacenar información sobre el producto y un constructor Producto(tipoProducto, hectareas) que crea un nuevo producto con el tipo de producto y las hectáreas especificadas. crearProductos(hectareasTotales): Este método crea una lista de productos a partir de las hectáreas totales disponibles. Toma como parámetro las hectáreas totales y devuelve la lista de productos creados.

validarHectareas(hectareasTotales): Este método valida el número de hectáreas ingresado por el usuario. Toma como parámetro las hectáreas totales y devuelve las hectáreas válidas ingresadas por el usuario.

imprimirTodos(productos): Este método imprime todos los productos de la lista. Toma como parámetro la lista de productos a imprimir.

imprimirElemento(producto): Este método imprime los atributos de un producto. Toma como parámetro el producto a imprimir.

actualizarPrecio(producto, precio): Este método actualiza el precio de un producto en la tabla de precios. Toma como parámetros el producto cuyo precio se actualizará y el nuevo precio del producto.

extraerProductos(productores): Este método extrae todos los productos de todos los productores. Toma como parámetro la lista de productores y devuelve una lista de todos los productos. getTipoProducto(): Este método devuelve el tipo de producto.

getCantidad(): Este método devuelve la cantidad de productos basándose en las hectáreas y el

rendimiento por hectárea. getHectareas(): Este método devuelve las hectáreas del producto.

getPrecio(fecha): Este método devuelve el precio del producto para una fecha especificada.

getPrecioConIVA(fecha): Este método devuelve el precio del producto con el IVA incluido para una fecha especificada.

getPesoTotal(): Este método devuelve el peso total de los productos.

NOMBRES_ATRIBUTOS: Esta es una constante que contiene los nombres de los atributos del producto para imprimir.

id, nombre, hectareas: Estos son campos de la clase Producto que representan el identificador, el nombre y las hectáreas del producto respectivamente.

productos: Este es un campo de la clase Productor que representa la lista de productos asociados al productor.

Productor(nombre, hectareas, productos): Este es el constructor de la clase Productor que crea

un nuevo productor con el nombre, las hectáreas y la lista de productos especificados.

crearProductor(): Este método estático crea un productor.

crearProductorFederado(productores, producto, productoresNombres, random): Este método estático crea un productor federado a partir de una lista de productores, un nombre de producto, una lista de nombres de productores y un indicador de aleatoriedad.

imprimirTodos(listaProductores): Este método imprime los detalles de todos los productores en la lista dada. Toma como parámetro la lista de productores a imprimir.
imprimirElemento(productor): Este método imprime los detalles del productor dado. Toma como parámetro el objeto Productor a imprimir.

borrarProductor(productores): Este método borra un productor de la lista de productores. Toma como parámetro la lista de productores y devuelve la lista actualizada después de eliminar el productor.

tipoProductor(productor): Este método devuelve el tipo de productor del productor dado. Toma como parámetro el objeto Productor para el cual se desea obtener el tipo y devuelve el tipo de productor. getNombre(): Este método devuelve el nombre del productor. getId(): Este método devuelve el ID del productor. getProductos(): Este método devuelve la lista de productos del productor.

setNombre(nombre): Este método establece el nombre del productor. Toma como parámetro el nuevo nombre del productor.

ProductorFactory: Esta clase se encarga de crear objetos de tipo Productor según los parámetros proporcionados.

crearProductor(nombre, hectareas, productos): Este método estático de ProductorFactory crea un objeto Productor según los parámetros especificados. Toma como parámetros el nombre del productor, las hectáreas del productor y la lista de productos del productor. Devuelve el objeto Productor creado.

ProductorFederado: Esta clase representa a un Productor Federado, que es una agrupación de Productores Pequeños.

ProductorFederado(nombre, hectareas, productos): Este es el constructor de la clase ProductorFederado que crea un objeto ProductorFederado con el nombre, las hectáreas y los productos especificados.

setProductorFederado(productores, productoresFederados, producto): Este método estático de ProductorFederado crea un objeto ProductorFederado a partir de una lista de Productores Pequeños y un Producto. Los Productores Pequeños se agrupan en un Productor Federado, sumando sus hectáreas y creando un nuevo Producto con las hectáreas totales. Toma como parámetros la lista de Productores Pequeños, la lista de Productores Federados existentes y el Producto que se utilizará para crear el nuevo Productor Federado. Devuelve el objeto ProductorFederado creado.

generarProductorRandom(isSmall): Este método estático genera un Productor aleatorio. Toma como parámetro isSmall, que indica si se debe generar un Productor pequeño. Devuelve el objeto Productor generado.

generarNombreRandom(): Este método estático genera un nombre aleatorio para un Productor. Devuelve el nombre generado.

generarHectareasRandom(): Este método estático genera un número aleatorio de hectáreas para un Productor. Devuelve el número de hectáreas generado.

ProductorGrande: Esta clase representa un Productor grande y extiende la clase abstracta Productor.

ProductorGrande(nombre, hectareas, productos): Este es el constructor de la clase ProductorGrande que crea un objeto ProductorGrande con el nombre, las hectáreas y los productos especificados.

ProductorPequeño: Esta clase representa un Productor pequeño y extiende la clase abstracta Productor.

MAX_PRODUCTOS: Esta constante define el número máximo de productos permitidos para un Productor pequeño.

ProductorPequeño(nombre, hectareas, productos): Este es el constructor de la clase ProductorPequeño que crea un objeto ProductorPequeño con el nombre, las hectáreas y los productos especificados.

getMaxProductos(): Este método estático devuelve el número máximo de productos permitidos para un Productor pequeño.

TablaPrecios: Esta clase se encarga de establecer y actualizar los precios derivados de productos.

precios: Este mapa almacena los precios actuales de los productos, donde la clave es el tipo de producto y el valor es el precio.

preciosAnteriores: Esta lista almacena los precios anteriores de los productos en diferentes fechas.

getPrecio(tipoProducto, fecha): Este método estático devuelve el precio del producto que se ajusta más a la fecha requerida. Toma como parámetros el tipo de producto y la fecha, y devuelve el precio.

setPrecio(precio, tipoProducto, fecha): Este método estático actualiza el precio de un tipo de producto. Toma como parámetros el nuevo precio, el tipo de producto y la fecha. No devuelve ningún valor.

productoTieneCambios(tipoProducto): Este método estático devuelve un booleano que indica si el precio del producto ha sufrido cambios. Toma como parámetro el tipo de producto y devuelve true si ha habido cambios, o false en caso contrario.

TipoComprador: Esta clase define los tipos de compradores disponibles y proporciona constantes estáticas para los tipos de comprador "Distribuidor" y "Consumidor final".

TipoProducto: Esta clase enumera los tipos de productos disponibles y sus atributos, como el id, nombre, si es perecedero, rendimiento por hectárea y peso. También contiene un array NOMBRES_ATRIBUTOS para imprimir.

getId(): Este método devuelve el id del tipo de producto. getNombre():

Este método devuelve el nombre del tipo de producto.

getEsPerecedero(): Este método devuelve un booleano que indica si el tipo de producto es perecedero.

getRendimientoHectarea(): Este método devuelve el rendimiento por hectárea de un tipo de producto.

getPeso(): Este método devuelve el peso de un producto

TipoProductor: Esta clase define los tipos de productores disponibles y proporciona constantes estáticas para los tipos de productor "grande", "pequeño" y "federado".

Utilidades: Esta clase de utilidades proporciona métodos útiles para la interacción con el usuario y la manipulación de objetos.

imprimirMenu(opciones): Este método estático imprime un menú en la consola con las opciones proporcionadas. Toma como parámetro un array de strings que representa las opciones del menú.

imprimirObjeto(objeto, atributos): Este método estático imprime los atributos de un objeto utilizando reflexión. Toma como parámetros el objeto del cual se imprimirán los atributos y un array de strings que representa los nombres de los atributos a imprimir.

getInt(mensaje): Este método estático lee un número entero desde la entrada estándar. Toma como parámetro un mensaje que se muestra al usuario y devuelve el número entero ingresado.

getDouble(mensaje, limit): Este método estático lee un número decimal desde la entrada estándar. Toma como parámetros un mensaje que se muestra al usuario y un límite máximo para el número decimal (0.0 si no hay límite). Devuelve el número decimal ingresado.

getString(mensaje): Este método estático lee una cadena de texto desde la entrada estándar. Toma como parámetro un mensaje que se muestra al usuario y devuelve la cadena de texto ingresada.

getBoolean(mensaje): Este método estático lee una respuesta booleana ("S" o "N") desde la entrada estándar. Toma como parámetro un mensaje que se muestra al usuario y devuelve true si la respuesta es "S", o false si la respuesta es "N".

4. Anexo con el código fuente de las clases implementadas.

```
/**

* Clase abstracta que representa a un Comprador. Proporciona funcionalidades

* para agregar productos a un pedido, crear un nuevo pedido, imprimir elementos

* de un comprador y mostrar información sobre los pedidos.

* usuario y la manipulación de objetos.
```

```
* @author (Sara Cubero)

* @version (1.0)

*/

import java.time.LocalDate; import java.util.ArrayList; import

java.util.HashSet; import java.util.List; import java.util.Set; public

abstract class Comprador { /* Constante para imprimir

compradores */ private static final String[]

NOMBRES_ATRIBUTOS = { "nombre" }; private String nombre;

private Pedido pedido;

/**

* Constructor de la clase Comprador.

*

* @param nombre nombre del comprador.

* @param pedido pedido del comprador.

*/

public Comprador(String nombre, Pedido pedido) {

this.nombre = nombre; this.pedido = pedido;


}

/**

* Extrae los nombres únicos de los productos de una lista de productos.

*

* @param productos la lista de productos.

* @return una lista que contiene los nombres únicos de los productos.

*/

private static List<String> extraerProductosUnicos(List<Producto> productos) {

Set<String> productosSet = new HashSet<>(); for (Producto producto :

productos) { productosSet.add(producto.getTipoProducto().getNombre());

}
```

```
return new ArrayList<>(productosSet);

}

/**

* Agrega productos a un pedido. Permite al usuario seleccionar el productor, el producto y la cantidad.

*

* @param productores lista de productores disponibles.

* @param kmCapital kilómetros a la capital.

* @param kmLocalidad kilómetros a la localidad.

* @param tipo tipo de comprador (1: Consumidor Final, 2: Distribuidor).

* @return una lista de elementos de pedido creados.

*/

public static List<ElementoPedido> agregarProductosPedido(List<Productor> productores, double
kmCapital, double kmLocalidad, int tipo, Productor productorElegido) {

List<ElementoPedido> elementosPedidos = new ArrayList<ElementoPedido>();

boolean opcion; do {


List<Producto> productosDisponibles = productorElegido.getProductos();

List<String> productosUnicos = extraerProductosUnicos(productosDisponibles);

System.out.println("Productos disponibles del productor " + productorElegido.getNombre() + ":"); for

(int i = 0; i < productosUnicos.size(); i++) {

System.out.println((i + 1) + ". " + productosUnicos.get(i));

}

int productoIndice = Utilidades.getInt("Introduzca el número del producto: ") - 1;

String productoNombre = productosUnicos.get(productoIndice); Producto

producto = null; int cantidadProducto = 0; for (Producto productoItem :

productosDisponibles) { if

(productoItem.getTipoProducto().getNombre().equals(productoNombre)) {

producto = productoItem; cantidadProducto += productoItem.getCantidad();

}

}
```

```
int cantidad = 0; int cantidadMaxima =

Integer.MAX_VALUE; int cantidadMinima =

1; if (tipo == 1) {

System.out.println("Cantidad máxima 100 kgr"); cantidadMaxima

= 100;

} else {

System.out.println("Cantidad mínima una tonelada"); cantidadMinima

= 1000;

}

System.out.println("Cantidad total de producto: " + cantidadProducto);

do { cantidad = Utilidades.getInt("Cantidad: ");


} while (cantidad > cantidadProducto || cantidad > cantidadMaxima || cantidad < cantidadMinima);

ElementoPedido elementoPedido = new ElementoPedido(cantidad, producto, kmCapital,

kmLocalidad);  elementosPedidos.add(elementoPedido); opcion = Utilidades.getBoolean("Agregar otro

producto:");

} while (opcion); return

elementosPedidos;

}

/**

* Crea un nuevo pedido a partir de una lista de elementos de pedido.

*

* @param elementosPedidos lista de elementos de pedido.

* @param tipo tipo de comprador (Consumidor Final o Distribuidor).

* @param nombre nombre del comprador.

* @param entrega días de entrega del pedido.

* @param productor productor.

* @return un objeto Comprador representando el pedido creado.

*/
```

```
public static Comprador crearPedido(List<ElementoPedido> elementosPedidos, String tipo, String nombre, int entrega, Productor productor) {
Pedido pedido = new Pedido(elementosPedidos, (LocalDate.now()).plusDays(entrega), productor);
Comprador comprador = CompradorFactory.crearComprador(tipo, nombre, pedido); return comprador;
}
/**
* Imprime los atributos de un comprador.
*
* @param comprador objeto Comprador.
*/
public static void imprimirElemento(Comprador comprador) {
Utilidades.imprimirObjeto(comprador, NOMBRES_ATRIBUTOS);
}
/**
* Imprime los compradores de un tipo específico (Consumidor Final o
* Distribuidor).
*
* @param compradores lista de compradores.
* @param esDistribuidor indica si se deben imprimir los compradores de tipo
* Distribuidor.
*/
public static void imprimirTipo(List<Comprador> compradores, boolean esDistribuidor) {
for (Comprador comprador : compradores) { if (esDistribuidor && comprador instanceof Distribuidor) {
Comprador.imprimirElemento(comprador);
} else if (!esDistribuidor && comprador instanceof ConsumidorFinal) {
Comprador.imprimirElemento(comprador);
}
```

```
}

}

/**

* Imprime los pedidos disponibles y muestra información detallada sobre un

* pedido seleccionado.

*

* @param compradores lista de compradores.

*/

public static void imprimirPedidos(List<Comprador> compradores) {

System.out.println("Pedidos disponibles:"); for (int i = 0; i <

compradores.size(); i++) {


System.out.println((i + 1) + ". " + compradores.get(i).getPedido().getId());

}

int pedidoIndice = Utilidades.getInt("¿Qué pedido imprimir? ") - 1;

Comprador comprador = compradores.get(pedidoIndice);

System.out.println("Comprador:" + comprador.getNombre());

boolean distribuidor = true; if (comprador instanceof

Distribuidor) {

System.out.println("Tipo: Distribuidor");

} else if (comprador instanceof ConsumidorFinal) {

System.out.println("Tipo: Consumidor Final"); distribuidor

= false;

}

System.out.println("Fecha:" + comprador.getPedido().getFecha());

System.out.println("Entrega:" + comprador.getPedido().getFechaEntrega());

System.out.println("Entregado: " + (comprador.getPedido().getEntregado() ? "sí" : "no"));

double totalKmGranLogistica = 0.0; double totalCosteGranLogistica = 0.0; double

totalKmPequeñaLogistica = 0.0; double totalCostePequeñaLogistica = 0.0; double
```

```
totalKmLogisticaPropia = 0.0; double totalCosteLogisticaPropia = 0.0; double
totalCosteLogistica = 0.0; for (ElementoPedido elemento :
comprador.getPedido().getElementos()) {
System.out.println("Producto: " + elemento.getProducto().getTipoProducto());
System.out.println("Cantidad: " + elemento.getCantidad()); for (Logistica
logistica : elemento.getLogistica()) { if (logistica instanceof GranLogistica) {
totalKmGranLogistica += logistica.getKm(); totalCosteGranLogistica +=
logistica.getCoste();

} else if (logistica instanceof PequeñaLogistica) {
totalKmPequeñaLogistica += logistica.getKm();
totalCostePequeñaLogistica += logistica.getCoste(); }
else if (logistica instanceof LogisticaPropia) {
totalKmLogisticaPropia += logistica.getKm();
totalCosteLogisticaPropia += logistica.getCoste();
}
totalCosteLogistica += logistica.getCoste();
}
if (totalKmGranLogistica > 0 || totalCosteGranLogistica > 0) {
System.out.println("Km totales Gran Logística: " + totalKmGranLogistica);
System.out.println("Coste total Gran Logística: " + totalCosteGranLogistica);
}
if (totalKmPequeñaLogistica > 0 || totalCostePequeñaLogistica > 0) {
System.out.println("Km totales Pequeña Logística: " + totalKmPequeñaLogistica);
System.out.println("Coste total Pequeña Logística: " + totalCostePequeñaLogistica);
}
if (totalKmLogisticaPropia > 0 || totalCosteLogisticaPropia > 0) {
System.out.println("Km totales Logística Propia: " + totalKmLogisticaPropia);
System.out.println("Coste total Logística Propia: " + totalCosteLogisticaPropia);
```

```
}

totalKmGranLogistica = 0.0;

totalCosteGranLogistica = 0.0;

totalKmPequeñaLogistica = 0.0;

totalCostePequeñaLogistica = 0.0;

totalKmLogisticaPropia = 0.0;

totalCosteLogisticaPropia = 0.0;

double precio = 0.0; double

beneficio = 0.0;


if (distribuidor) {

beneficio = (elemento.getProducto().getPrecio(comprador.getPedido().getFechaEntrega()) *
PorcentajeTipoComprador.PORCENTAJE_DISTRIBUIDOR) / 100; precio =
elemento.getProducto().getPrecio(comprador.getPedido().getFechaEntrega())

* elemento.getCantidad();

} else {

beneficio = (elemento.getProducto().getPrecio(comprador.getPedido().getFechaEntrega()) *
PorcentajeTipoComprador.PORCENTAJE_CONSUMIDOR_FINAL) / 100; precio =
elemento.getProducto().getPrecioIVA(comprador.getPedido().getFechaEntrega()) *
elemento.getCantidad();

}

System.out.println("Beneficio de cooperativa:" + beneficio);

System.out.println("Precio total con logistica:" + (precio + beneficio + totalCosteLogistica));

}

}

/**

* Obtiene el pedido asociado a un comprador.

*

* @return el objeto Pedido asociado al comprador.

*/
```

```
public Pedido getPedido() { return

pedido;

}

/**

* Obtiene el nombre del comprador.

*

* @return el nombre del comprador.

*/

public String getNombre() { return

nombre;


}

}

/**

* Fábrica para crear objetos Comprador. Proporciona un método estático para

* crear diferentes tipos de compradores según el tipo especificado.

* @author (Sara Cubero)

* @version (1.0)

*/

public class CompradorFactory {

/**

* Crea un objeto Comprador según el tipo especificado.

*

* @param tipo tipo de comprador (DISTRIBUIDOR o CONSUMIDOR_FINAL).

* @param nombre nombre del comprador.

* @param pedido pedido asociado al comprador.

* @return un objeto Comprador creado según el tipo especificado.

*/
```

```
public static Comprador crearComprador(String tipo, String nombre, Pedido pedido) {
if (tipo.equalsIgnoreCase(TipoComprador.DISTRIBUIDOR)) { return new
Distribuidor(nombre, pedido);
} else if (tipo.equalsIgnoreCase(TipoComprador.CONSUMIDOR_FINAL)) { return
new ConsumidorFinal(nombre, pedido);
} else { return
null;
}
}
}
/**
* Generador de compradores aleatorios. Proporciona un método estático para
* generar una lista de compradores con información aleatoria, como tipo de
* comprador, nombre y elementos de pedido.
* @author (Sara Cubero)
* @version (1.0)
*/
import java.util.ArrayList; import java.util.List; import
java.util.Random; public class CompradorGenerator {
/* Constante random */ private static final Random
random = new Random();
/**
* Genera una lista de compradores aleatorios basados en la lista de productos
* proporcionada.
*
* @param productores lista de productores disponibles.
* @return una lista de compradores generados aleatoriamente.
*/
```

```
public static List<Comprador> generarCompradores(List<Productor> productores) { List<Comprador>
compradores = new ArrayList<>();
for (int i = 0; i < 10; i++) {
String tipo = generarTipoComprador();
String nombre = generarNombreComprador(); Productor productor =
productores.get(i); for (int j = i; j < productores.size(); j++) { if(
productores.get(j) instanceof ProductorFederado ) { if(
productores.get(j).getNombre().contains(productores.get(i).getNombre()) ) {
productor = productores.get(j); break;

}
}
}
List<ElementoPedido> elementosPedido = agregarElementosPedido(productor.getProductos());
Comprador comprador = Comprador.crearPedido(elementosPedido, tipo, nombre, 10, productor);
compradores.add(comprador);
}
return compradores;
}
/**
* Agrega elementos de pedido aleatorios a una lista de elementos de pedido.
*
* @param productos lista de productos disponibles.
* @return una lista de elementos de pedido generados aleatoriamente.
*/
private static List<ElementoPedido> agregarElementosPedido(List<Producto> productos) {
List<ElementoPedido> elementosPedido = new ArrayList<>(); if (!productos.isEmpty()) { int
numProductos = random.nextInt(productos.size()) + 1; for (int i = 0; i < numProductos; i++)
{
```

```
Producto producto = productos.get(random.nextInt(productos.size())); int cantidad =
random.nextInt(10) + 1; double kmCapital = Math.round(random.nextDouble() * 100);
double kmLocalidad = Math.round(random.nextDouble() * 50); elementosPedido.add(new
ElementoPedido(cantidad, producto, kmCapital, kmLocalidad));
}
}
return elementosPedido;
}
/**
* Genera un tipo de comprador aleatorio (DISTRIBUIDOR o CONSUMIDOR_FINAL).
*
* @return el tipo de comprador generado aleatoriamente.
*/
private static String generarTipoComprador() {
if (random.nextBoolean()) { return
TipoComprador.DISTRIBUIDOR;
} else {
return TipoComprador.CONSUMIDOR_FINAL;
}
}
/**
* Genera un nombre de comprador aleatorio utilizando una combinación de nombres
* y apellidos.
*
* @return el nombre de comprador generado aleatoriamente.
*/
private static String generarNombreComprador() {
String[] nombres = { "Juan", "Pedro", "Luis", "Ana", "María", "Lucía", "Miguel", "José", "Carla", "Elena",
"Pablo", "Alberto" };
```

```
String[] apellidos = { "García", "Rodríguez", "Martínez", "López", "González", "Pérez", "Sánchez",
"Ramírez",

"Torres", "Flores", "Vargas", "Castillo" }; int indexNombre =

new Random().nextInt(nombres.length); int indexApellido =

new Random().nextInt(apellidos.length); return

nombres[indexNombre] + " " + apellidos[indexApellido];

}

}

/**

* Clase que representa a un Comprador de tipo Consumidor Final.

* Hereda de la clase Comprador y proporciona un constructor para crear un objeto ConsumidorFinal.

* @author (Sara Cubero)

* @version (1.0)

*/

/**

* Clase que representa a un Comprador de tipo Consumidor Final.

* Hereda de la clase Comprador y proporciona un constructor para crear un objeto ConsumidorFinal.

*/

public class ConsumidorFinal extends Comprador {

/**

* Constructor de la clase ConsumidorFinal.

* @param nombre nombre del comprador.

* @param pedido pedido del comprador.

*/

public ConsumidorFinal(String nombre, Pedido pedido) { super(nombre,

pedido);

}

}

/**
```

```
* Esta clase representa una Cooperativa con productores y compradores.
* @author (Sara Cubero)
* @version (1.0)
*/
import java.time.LocalDate; import
java.time.format.DateTimeFormatter; import
java.util.ArrayList; import java.util.HashSet;

import java.util.Iterator;
import java.util.List; import
java.util.Set;
/**
* Esta clase representa una Cooperativa con productores y compradores.
*/
public class Cooperativa {
// Mensajes constantes usados en la aplicación.
private static final String AÑADIR_OPCION = "Ingrese una opción:"; private
static final String OPCION_INVALIDA = "Opción inválida.";
// Listas para almacenar los productores y compradores.
private static List<Productor> productores = new ArrayList<Productor>(); private
static List<Comprador> compradores = new ArrayList<Comprador>();
/**
* Punto de entrada principal de la aplicación.
*
* @param args argumentos de la línea de comandos.
*/
public static void main(String[] args) {
cargaPrincipal(); menuPrincipal();
}
```

```
/**
* Este método muestra el menú principal y maneja la entrada del usuario.
*/
private static void menuPrincipal() {
int opcion; do { try {
Utilidades.imprimirMenu(Menus.ME
NU_PRINCIPAL); opcion =
Utilidades.getInt(AÑADIR_OPCION);
switch (opcion) { case 0: break;
case 1: menuProductor(); break;
case 2: menuCompradores(); break;
case 3: menuEstadistica(); break;
default:
System.out.println(OPCION_INVALIDA);
}
} catch (Exception e) {
System.out.println("Ha habido un error: " + e.getMessage()); opcion
= -1;
}
} while (opcion != 0);
}
/**
* Presenta un menú para interactuar con objetos Productor. Permite al usuario
* elegir opciones como crear un nuevo Productor, imprimir todos los * Productores, eliminar un Productor,
  establecer un límite para las hectáreas y
* actualizar el precio de un producto.
*/
private static void menuProductor() {
int opcion; do {
```

```
Utilidades.imprimirMenu(Menus.MENU_PRODUCTORES);

opcion = Utilidades.getInt(AÑADIR_OPCION); switch

(opcion) { case 0: break; case 1: menuCrearProductor();

break; case 2:

Productor.imprimirTodos(productores);

break; case 3:

Productor.borrarProductor(productores);

break; case 4: menuLimiteHectareas();

break; case 5:

Set<String> productosUnicos = new HashSet<>();

List<Producto> productos = Producto.extraerProductos(productores); for

(Producto producto : productos) {

productosUnicos.add(producto.getTipoProducto().getNombre());

}

System.out.println("Productos disponibles:");

int i = 1; for (String productoUnico :

productosUnicos) { System.out.println(i + ". " +

productoUnico); i++;

}

int opcionProducto = Utilidades.getInt("Ingrese el número del producto que desea actualizar: "); String

productoNombre = "";

i = 1; for (String productoUnico :

productosUnicos) { if (i == opcionProducto) {

productoNombre = productoUnico; break;

}

i++;

}

double precio = Utilidades.getDouble("¿Qué precio desea poner?", 0.0); for

(Producto producto : productos) { if
```

```
(producto.getTipoProducto().getNombre().equalsIgnoreCase(productoNombre)) {
Producto.actualizarPrecio(producto, precio); return;
}
}
System.out.println("No se ha encontrado ningún producto con ese nombre.");
break; default:
System.out.println(OPCION_INVALIDA);
}
System.out.println();
} while (opcion != 0);
}
/**
* Presenta un menú para crear nuevos objetos de Productor. El usuario puede
* elegir entre crear un Productor normal o crear un Productor Federado.
*/
private static void menuCrearProductor() {
int opcion; do {
Utilidades.imprimirMenu(Menus.MENU_CREAR_PRODUCTORES);
opcion = Utilidades.getInt(AÑADIR_OPCION); switch (opcion) {
case 0: break; case 1:
productores.add(Productor.crearProductor()); break; case 2:
crearProductorFederado(); break; default:
System.out.println(OPCION_INVALIDA);
}
System.out.println();
} while (opcion != 0);
}
/**
* Crea un nuevo Productor Federado a partir de varios Productores Pequeños. El
```

```
* usuario puede seleccionar los productores y el producto que se federará.
*/
private static void crearProductorFederado() {
List<Productor> productoresPequeños = new ArrayList<>();
Set<String> nombresProductosUnicos = new HashSet<>(); for
(Productor productor : productores) { if (productor instanceof
ProductorPequeño) { productoresPequeños.add(productor); for
(Producto producto : productor.getProductos()) {
nombresProductosUnicos.add(producto.getTipoProducto().getNombre());
}
}
}
List<String> nombresProductosLista = new ArrayList<>(nombresProductosUnicos); for
(int i = 0; i < nombresProductosLista.size(); i++) {
System.out.println((i + 1) + "- " + nombresProductosLista.get(i));
}
int indiceProducto = Utilidades.getInt("¿Qué producto desea federar? (Ingrese el número)") - 1;
String producto = nombresProductosLista.get(indiceProducto); boolean otro = false;
List<String> productoresNombres = new ArrayList<String>();
Iterator<Productor> iterator = productoresPequeños.iterator();
while (iterator.hasNext()) { Productor productor =
iterator.next(); boolean tieneProducto = false; for (Producto
prod : productor.getProductos()) { if
(prod.getTipoProducto().getNombre().equals(producto)) {
tieneProducto = true; break;
}
}
if (!tieneProducto) { iterator.remove();
```

```
}

}

if (productoresPequeños.isEmpty()) { throw new

IllegalArgumentException("El producto ingresado es incorrecto.");

} else { do { for (int i = 0; i <

productoresPequeños.size(); i++) {

System.out.println((i + 1) + "- " + productoresPequeños.get(i).getNombre());

}

int indiceElegido = Utilidades.getInt("¿De qué productor? (Ingrese el número)") - 1;

if (indiceElegido >= 0 && indiceElegido < productoresPequeños.size()) { Productor

productorElegido = productoresPequeños.get(indiceElegido);

productoresNombres.add(productorElegido.getNombre());

productoresPequeños.remove(indiceElegido);

} else {

System.out.println("?ndice de productor inválido.");

}

if (productoresPequeños.size() == 0) { otro = false; } else

{ otro = Utilidades.getBoolean("¿Agregar otro

productor?");

}

} while (otro);

}

Productor productorFederado = Productor.crearProductorFederado(productores, producto,
productoresNombres,

false); for (Productor productorItem :

productores) { if (productorItem instanceof

ProductorPequeño) { ProductorPequeño

productorPequeño = (ProductorPequeño)
```

```
productorItem; if
(productorPequeño.getProductos().stream()
.anyMatch(p -> p.getTipoProducto().getNombre().equals(producto))) {
for (Producto productoItem : productorPequeño.getProductos()) { if
(productoItem.getTipoProducto().getNombre().equals(producto)) {
productorPequeño.getProductos().remove(productoItem); break;
}
}
}
}
}
productores.add(productorFederado);
}
/**
* Presenta un menú para establecer un límite en las hectáreas. El usuario puede
* cambiar el valor del límite.
*/
private static void menuLimiteHectareas() {
int opcion; do {
LimiteHectareas limite = LimiteHectareas.getInstance(); double
limiteHectareas = limite.getLimite();
String[] opcionesMenu = { "LÍMITE HECTAREAS",
"1. Cambiar valor",
"0. Volver",
"Valor actual:".concat(String.valueOf(limiteHectareas)), };
Utilidades.imprimirMenu(opcionesMenu);      opcion      =
Utilidades.getInt(AÑADIR_OPCION);
```

```
switch (opcion) {
case 0: break;
case 1:
double valor = Utilidades.getDouble("Ingrese un valor: ", 0.0); if
(valor <= 0) {
System.out.print("No es un valor válido");
}
limite.setLimite(valor);
List<Productor> productoresAux = new ArrayList<Productor>(); for (Productor productor : productores) {
productoresAux.add(ProductorFactory.crearProductor(productor.getNombre(), productor.getHectareas(),
productor.getProductos()));
}
productores = productoresAux;
break; default:
System.out.println(OPCION_INVALIDA);
}
System.out.println();
} while (opcion != 0);
}
/**
* Presenta un menú para interactuar con objetos Comprador. Permite al usuario * elegir
  opciones como imprimir Compradores, imprimir Pedidos, crear un nuevo
* Pedido, y verificar las entregas y precios.
*/
private static void menuCompradores() { int
opcion;

do {
```

```
Utilidades.imprimirMenu(Menus.MENU_COMPRADORES);

opcion = Utilidades.getInt(AÑADIR_OPCION); switch

(opcion) { case 0: break; case 1:

Comprador.imprimirTipo(compradores,  true);

break; case 2:

Comprador.imprimirTipo(compradores,  false);

break; case 3:

Comprador.imprimirPedidos(compradores);

break; case 4:

int tipo = Utilidades.getInt("Tipo de comprador: 1- Consumidor Final 2-Distribuidor");

String nombre = Utilidades.getString("Nombre de comprador: "); double kmCapital =

Utilidades.getDouble("KM a capital: ", 0.0); double kmLocalidad =

Utilidades.getDouble("KM a localidad: ", 100.0); int entrega = Utilidades.getInt("Días de

envio (estándar - 10): "); System.out.println("Productores disponibles:"); for (int i = 0; i <

productores.size(); i++) { boolean tieneFederacion = false; for (int j = i + 1; j <

productores.size(); j++) { if( productores.get(j) instanceof ProductorFederado ) { if(

productores.get(j).getNombre().contains(productores.get(i).getNombre()) ) { if(

productores.get(j).getProductos().size() == productores.get(i).getProductos().size() ) {

boolean esta = false;


for( Producto productoI : productores.get(i).getProductos() ) { for( Producto productoJ :

productores.get(j).getProductos() ) { if(

productoI.getTipoProducto().getNombre().equals(productoJ.getTipoProducto().getNombre()) ) {

esta = true;

} } if( !esta

) {

break;

} } if( esta

) {
```

```
tieneFederacion = true;
}
}
}
}
}
if( !tieneFederacion ) {
System.out.println((i + 1) + ". " + productores.get(i).getNombre());
}
}
int productorIndice = Utilidades.getInt("Introduzca el número del productor: ") - 1; Productor
productorElegido = productores.get(productorIndice);
compradores.add(Comprador.crearPedido(
Comprador.agregarProductosPedido(productores, kmCapital, kmLocalidad, tipo, productorElegido), (tipo
== 1) ? TipoComprador.CONSUMIDOR_FINAL : TipoComprador.DISTRIBUIDOR, nombre, entrega,
productorElegido)); break; case 5:

verificarEntregasYPrecios(null, true);
break; default:
System.out.println(OPCION_INVALIDA);
}
System.out.println();
} while (opcion != 0);
}
/**
* Presenta un menú para interactuar con las estadísticas. Permite al usuario
* elegir opciones como visualizar las ventas totales, los ingresos obtenidos
* por productor, los ingresos obtenidos por logística, los beneficios de la
* cooperativa por producto, y la evolución de los precios del producto.
```

```
*/
private static void menuEstadistica() {
int opcion; do {
Utilidades.imprimirMenu(Menus.MENU_ESTADISTICA);
opcion = Utilidades.getInt(AÑADIR_OPCION); switch
(opcion) { case 0: break; case 1:
Estadistica.ventasTotales(Pedido.extraerPedidos(compradores));
break; case 2:
Estadistica.importesObtenidosProductor(Pedido.extraerPedidos(compradores), productores);
break; case 3:
Estadistica.importesObtenidosLogistica(Pedido.extraerPedidos(compradores));
break; case 4:
Estadistica.beneficiosCooperativaPorProducto(compradores);
break; case 5:
Estadistica.evolucionPreciosProducto(Producto.extraerProductos(productores));
break; default:
System.out.println(OPCION_INVALIDA);
}
System.out.println();
} while (opcion != 0);
}
/**
* Carga los datos iniciales de la aplicación. Crea productores y compradores de
* prueba y actualiza los precios de los productos.
*/
private static void cargaPrincipal() { for
(int i = 0; i < 5; i++) {
productores.add(ProductorGenerator.generarProductorRandom(false));
```

```
} for (int i = 0; i < 5; i++)
{
productores.add(ProductorGenerator.generarProductorRandom(true));
}
Productor duplicado = new ProductorPequeño("Curro", productores.get(9).getHectareas(),
productores.get(9).getProductos());
List<String> productoresNombres = new ArrayList<String>();
productoresNombres.add(duplicado.getNombre());
productoresNombres.add(productores.get(9).getNombre()); productores.add(duplicado);
productores.add(Productor.crearProductorFederado(productores,
duplicado.getProductos().get(0).getTipoProducto().getNombre(), productoresNombres, true));
compradores = CompradorGenerator.generarCompradores(productores);
PrecioGenerator.actualizarPreciosUnicos(Producto.extraerProductos(productores));
verificarEntregasYPrecios(LocalDate.now().plusDays(10), false);
}
/**
* Verifica las entregas y precios en una fecha determinada. Si se proporciona * una fecha,
  verifica las entregas y precios en esa fecha. Si no se proporciona
* una fecha, verifica las entregas y precios en la fecha actual.
*/
private static void verificarEntregasYPrecios(LocalDate fechaAutomatica, boolean imprimir) {
LocalDate fecha = fechaAutomatica; if (fechaAutomatica == null) {
String fechaString = Utilidades.getString("Por favor, ingrese una fecha en el formato YYYY-MM-DD:");
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd"); fecha =
LocalDate.parse(fechaString, formatter);
}
for (Comprador comprador : compradores) { Pedido pedido =
comprador.getPedido(); if (pedido.getFechaEntrega().compareTo(fecha) <= 0 &&
!pedido.getEntregado()) { pedido.setEntregado(true);
```

```
if (imprimir) {
System.out.println("Pedido entregado:" + pedido.getId());
}
}
}
}
}
/**
* Clase que representa a un Comprador de tipo Distribuidor.
* Hereda de la clase Comprador y proporciona un constructor para crear un objeto Distribuidor.
* @author (Sara Cubero)
* @version (1.0)
*/
public class Distribuidor extends Comprador {
/**
* Constructor de la clase Distribuidor.
* @param nombre nombre del comprador.
* @param pedido pedido del comprador.
*/
public Distribuidor(String nombre, Pedido pedido) { super(nombre,
pedido);
}
}
/**
* Clase que representa un elemento de un pedido.
* @author (Sara Cubero)
* @version (1.0)
*/
```

```
import java.util.List; public

class ElementoPedido {

/** La cantidad del producto en el pedido */

private int cantidad; /** El producto del

elemento */ private Producto producto;


/** La lista de logísticas asociadas al elemento */ private

List<Logistica> logisticas;

/**

* Constructor de la clase ElementoPedido.

*

* @param cantidad La cantidad del producto en el pedido.

* @param producto El producto del elemento.

* @param kmCapital Los kilómetros hacia la capital.

* @param kmLocalidad Los kilómetros hacia la localidad.

*/

public ElementoPedido(int cantidad, Producto producto, double kmCapital, double kmLocalidad) {

this.cantidad = cantidad; this.producto = producto; logisticas = Logistica.generarEnvio(producto,

kmCapital, kmLocalidad, cantidad);

}

/**

* Obtiene el nombre del producto del elemento.

*

* @return El nombre del producto.

*/

public String getProductoNombre() { return

producto.getTipoProducto().getNombre();

}

/**
```

```
* Obtiene la cantidad del producto en el elemento.
*
* @return La cantidad del producto.
*/
public int getCantidad() {
return cantidad;
}
/**
* Obtiene el peso total del elemento.
*
* @return El peso total del elemento.
*/
public double getPeso() { return cantidad *
producto.getTipoProducto().getPeso();
}
/**
* Obtiene el producto del elemento.
*
* @return El producto del elemento.
*/
public Producto getProducto() { return
producto;
}
/**
* Obtiene la lista de logísticas asociadas al elemento.
*
* @return La lista de logísticas.
*/
```

```
public List<Logistica> getLogistica() { return

logisticas;

}

/**

* Establece la cantidad del producto en el elemento.

*

* @param cantidad La nueva cantidad del producto.

*/

public void setCantidad(int cantidad) { this.cantidad

= cantidad;

}

}

/**

* Clase que contiene métodos para calcular diversas estadísticas relacionadas

* con los pedidos, ventas y precios.

* @author (Sara Cubero)

* @version (1.0)

*/

import java.time.LocalDate;

import java.util.HashMap;

import java.util.HashSet;

import java.util.List; import

java.util.Map; import

java.util.Set; import

java.util.regex.Matcher; import

java.util.regex.Pattern; public

class Estadistica {

/**

* Calcula las ventas totales por producto en un período de tiempo específico.
```

```
*
* @param pedidos la lista de pedidos a analizar
*/
public static void ventasTotales(List<Pedido> pedidos) {
LocalDate fechaInicio, fechaFin; do {

String inicio = Utilidades.getString("Introduzca la fecha de inicio (formato YYYY-MM-DD):");
String fin = Utilidades.getString("Introduzca la fecha final (formato YYYY-MM-DD):");
fechaInicio = LocalDate.parse(inicio); fechaFin = LocalDate.parse(fin);
} while (fechaInicio.isAfter(fechaFin));
Map<String, Integer> ventasPorProducto = new HashMap<>(); for (Pedido pedido : pedidos) { if
(!pedido.getFechaEntrega().isBefore(fechaInicio) && !pedido.getFechaEntrega().isAfter(fechaFin)) {
for (ElementoPedido elemento : pedido.getElementos()) {
String nombreProducto = elemento.getProducto().getTipoProducto().getNombre(); int ventas =
elemento.getCantidad(); ventasPorProducto.put(nombreProducto,
ventasPorProducto.getOrDefault(nombreProducto, 0) + ventas);
}
}
}
for (Map.Entry<String, Integer> entry : ventasPorProducto.entrySet()) {
System.out.println("Producto: " + entry.getKey() + ", Ventas: " + entry.getValue());
}
if (ventasPorProducto.isEmpty()) {
System.out.println("No se ha vendido ningun producto en ese período");
}
}
/**
* Calcula los ingresos obtenidos por cada productor en función de los pedidos entregados.
*
```

```
* @param pedidos la lista de pedidos a analizar.

* @param lista de productores para buscar los productores si hay un pedido a una federacion

*/

public static void importesObtenidosProductor(List<Pedido> pedidos, List<Productor> productores) {

Map<String, Map<String, Double>> ingresosPorProductoPorProductor = new HashMap<>(); for

(Pedido pedido : pedidos) { if (pedido.getEntregado()) {

Productor productor = pedido.getProductor();

if( productor instanceof ProductorFederado ) {

double total = 0; for (int i = 0; i <

productores.size(); i++) {

if( productor.getNombre().contains(productores.get(i).getNombre()) &&
!productor.getNombre().equals(productores.get(i).getNombre()) ) {

Map<String, Double> ingresosPorProducto =
ingresosPorProductoPorProductor.getOrDefault(productores.get(i).getNombre(), new HashMap<>());

String stringPattern = "\\((.*?)\\)";

Pattern pattern = Pattern.compile(stringPattern); if(

total == 0 ) {

String[] hectareasProductores = productor.getNombre().split("-");

for( String hectareasProductor : hectareasProductores ) { Matcher

match = pattern.matcher(hectareasProductor);

if (match.find()) { total +=

Double.parseDouble(match.group(1));

}

}

}

String hectareasProductor = productor.getNombre().split(productores.get(i).getNombre())[1];

Matcher match = pattern.matcher(hectareasProductor);

Double hectareas = 1.0; if (match.find()) { hectareas =

Double.parseDouble(match.group(1));

}
```

```
for (ElementoPedido elemento : pedido.getElementos()) {

double ingreso = ( ( elemento.getProducto().getPrecio(pedido.getFechaEntrega()) *
elemento.getCantidad() ) * hectareas ) / total;

String nombreProducto = elemento.getProducto().getTipoProducto().getNombre();

ingresosPorProducto.put(nombreProducto, ingresosPorProducto.getOrDefault(nombreProducto, 0.0) +
ingreso);

}

ingresosPorProductoPorProductor.put(productores.get(i).getNombre(), ingresosPorProducto);

}

}

} else {

Map<String, Double> ingresosPorProducto =
ingresosPorProductoPorProductor.getOrDefault(productor.getNombre(), new HashMap<>()); for

(ElementoPedido elemento : pedido.getElementos()) { double ingreso =

elemento.getProducto().getPrecio(pedido.getFechaEntrega()) * elemento.getCantidad(); String

nombreProducto = elemento.getProducto().getTipoProducto().getNombre();

ingresosPorProducto.put(nombreProducto, ingresosPorProducto.getOrDefault(nombreProducto, 0.0) +
ingreso);

}

ingresosPorProductoPorProductor.put(productor.getNombre(), ingresosPorProducto);

}

}

}

for (Map.Entry<String, Map<String, Double>> entryProductor :
ingresosPorProductoPorProductor.entrySet()) {

String nombreProductor = entryProductor.getKey(); System.out.println("Productor: " +

nombreProductor); for (Map.Entry<String, Double> entryProducto :

entryProductor.getValue().entrySet()) {

System.out.println("Producto: " + entryProducto.getKey() + ", Ingreso obtenido: " +
entryProducto.getValue());

}

}
```

```
}

/**

* Calcula los importes obtenidos por cada tipo de logística en función de los

* pedidos entregados.

*

* @param pedidos la lista de pedidos a analizar

*/

public static void importesObtenidosLogistica(List<Pedido> pedidos) {

Map<String, Double> costesPorLogistica = new HashMap<>(); for

(Pedido pedido : pedidos) { if (pedido.getEntregado()) { for

(ElementoPedido elemento : pedido.getElementos()) { for (Logistica

logistica : elemento.getLogistica()) { double coste =

logistica.getCoste();

String nombreClaseLogistica = logistica.getClass().getSimpleName();

costesPorLogistica.put(nombreClaseLogistica,

costesPorLogistica.getOrDefault(nombreClaseLogistica, 0.0) + coste);

}

}

}

}

for (Map.Entry<String, Double> entry : costesPorLogistica.entrySet()) {

System.out.println("Logística: " + entry.getKey() + ", Importe obtenido: " + entry.getValue());

}

}

/**

* Calcula los beneficios totales obtenidos por la cooperativa por cada

* producto.

*

* @param compradores la lista de compradores
```

```
*/
public static void beneficiosCooperativaPorProducto(List<Comprador> compradores) { Map<String,
Double> beneficiosPorProducto = new HashMap<>(); for (Comprador comprador : compradores) { if
(comprador.getPedido().getEntregado()) { for (ElementoPedido elemento :
comprador.getPedido().getElementos()) { double beneficio = 0.0; if (comprador instanceof
Distribuidor) { beneficio =
(elemento.getProducto().getPrecio(comprador.getPedido().getFechaEntrega()) * 5) / 100;
} else if (comprador instanceof ConsumidorFinal) { beneficio =
(elemento.getProducto().getPrecio(comprador.getPedido().getFechaEntrega()) * 15) / 100;
}
beneficio *= elemento.getCantidad();
String tipoProducto = elemento.getProducto().getTipoProducto().getNombre();
beneficiosPorProducto.put(tipoProducto, beneficiosPorProducto.getOrDefault(tipoProducto,
0.0) + beneficio);
}
}
for (Map.Entry<String, Double> entry : beneficiosPorProducto.entrySet()) {
System.out.println("Producto: " + entry.getKey() + ", Beneficio: " + entry.getValue());
}
}
}
/**
* Calcula la evolución de precios de cada producto.
*
* @param productos la lista de productos a analizar
*/
public static void evolucionPreciosProducto(List<Producto> productos) {
boolean cambiosEnPrecios = false;
```

```
Set<TipoProducto> productosImpresos = new HashSet<>(); for (Producto producto :
productos) { if (!productosImpresos.contains(producto.getTipoProducto())) { double
precioActual = TablaPrecios.getPrecio(producto.getTipoProducto(), LocalDate.now());
System.out.println("Precio actual de " + producto.getTipoProducto().getNombre() + ": " + precioActual);
List<Map<LocalDate, Map<TipoProducto, Double>>> preciosAnteriores = TablaPrecios
.getPreciosAnteriores(producto.getTipoProducto());
if (!preciosAnteriores.isEmpty()) { cambiosEnPrecios
= true;
System.out.println("Evolución de precios para " + producto.getTipoProducto().getNombre() + ":"); for
(Map<LocalDate, Map<TipoProducto, Double>> registroPrecios : preciosAnteriores) {
LocalDate fecha = registroPrecios.keySet().iterator().next();
Map<TipoProducto, Double> preciosMap = registroPrecios.get(fecha);
double precioAnterior = preciosMap.get(producto.getTipoProducto()); if
(precioAnterior != precioActual) {
System.out.println("Fecha: " + fecha + ", Precio:" + precioAnterior);
}
precioActual = precioAnterior;
}
}
System.out.println(); productosImpresos.add(producto.getTipoProducto());
}
}
if (!cambiosEnPrecios) {
System.out.println("No se produjeron cambios en los precios.");
}
}
}
/**
* Clase que representa una logística de gran tamaño para el transporte de productos.
```

```
* Hereda de la clase Logistica.

* @author (Sara Cubero)

* @version (1.0)

*/

public class GranLogistica extends Logistica { private

static final double PRECIO_KM = 0.05;

/**

* Constructor de la clase GranLogistica.

* @param coste el coste de la logística

* @param kg el peso en kg de la logística

* @param km la distancia en km de la logística

* @param producto el producto asociado a la logística

*/

public GranLogistica(double coste, double kg, double km) { super(coste,

kg, km);

}

/**

* Obtiene el precio por kilómetro de la logística de gran tamaño.

* @return el precio por kilómetro

*/

public static double getPrecioKm() { return

PRECIO_KM;

}

}

/**

* Esta clase representa el límite máximo de hectáreas que un productor puede tener en la cooperativa
  empresarial.

* Esta clase es implementada como un patrón Singleton para garantizar que solo exista una instancia del
  límite de hectáreas.

* @author (Sara Cubero)
```

```
* @version (1.0)
*/
public class LimiteHectareas {
/**
* Límite de hectáreas
*/
private double limite;
/**
* Instancia única de la clase
*/
private static LimiteHectareas instance = null;
/**
* Constante para el límite por defecto
*/
private final static double LIMITE_DEFAULT = 5;
private LimiteHectareas(double limite) { this.limite
= limite;
}
/*
* Método para obtener la instancia única de la clase
*/
public static LimiteHectareas getInstance() { if
(instance == null) { instance = new
LimiteHectareas(LIMITE_DEFAULT);
}
return instance;
}
/**
* Método para establecer un límite nuevo
```

```
*
* @param limite (double)
*/
public void setLimite(double limite) { this.limite
= limite;
}
/**
* Método para obtener el límite actual
*
* @return limite (double)
*/
public double getLimite() {
return this.limite;
}
}
/**
* Clase abstracta que representa una entidad de logística. Esta clase sirve
* como base para implementar diferentes tipos de logística.
* @author (Sara Cubero)
* @version (1.0)
*/
import java.util.List; public
abstract class Logistica {
private double coste; private
double km;

private double kg;
/**
* Constructor de la clase Logistica.
```

```
*
* @param coste El coste de la logística.
* @param kg Los kilogramos de carga de la logística.
* @param km Los kilómetros recorridos en la logística.
*/
public Logistica(double coste, double kg, double km) {
this.coste = coste; this.kg = kg; this.km = km;
}
/**
* Método estático para generar un envío de logística. Este método utiliza la
* factoría de logística para crear el envío.
*
* @param producto El producto a enviar.
* @param kmCapital Los kilómetros en ruta hacia la capital.
* @param kmLocalidad Los kilómetros en ruta hacia la localidad.
* @param cantidad La cantidad de productos a enviar.
* @return Una lista de objetos de tipo Logistica que representan el envío.
*/
public static List<Logistica> generarEnvio(Producto producto, double kmCapital, double kmLocalidad,
int cantidad) { return LogisticaFactory.crearLogistica(kmCapital, kmLocalidad, producto, cantidad);
}
/**
* Obtiene los kilómetros recorridos en la logística.
*
* @return Los kilómetros recorridos.
*/
public double getKm() {
return km;
}
```

```
/**
* Obtiene los kilogramos de carga de la logística.
*
* @return Los kilogramos de carga.
*/
public double getKg() { return
kg;
}
/**
* Obtiene el coste de la logística.
*
* @return El coste de la logística.
*/
public double getCoste() { return
coste;
}
/**
* Obtiene el tipo de logística.
*
* @return El tipo de logística como una cadena de texto.
*/
public String getTipo() { return
this.getClass().getSimpleName(); }

}
/**
* Clase que se encarga de crear objetos de logística según los parámetros
* proporcionados.
* @author (Sara Cubero)
```

```
* @version (1.0)
*/
import java.time.LocalDate; import
java.util.ArrayList; import java.util.List; import
java.util.Map.Entry; public class
LogisticaFactory { /* Constantes valores peso y
kms */ private static final int KMS_MAXIMOS
= 100; private static final int KMS_TRAMO =
50; private static final int PESO_MAXIMO =
1000;
/**
* Crea una lista de objetos de logística según los parámetros especificados.
*
* @param kmCapital Los kilómetros en ruta hacia la capital.
* @param kmLocalidad Los kilómetros en ruta hacia la localidad.
* @param producto El producto a enviar.
* @param cantidad La cantidad de productos a enviar.
* @return Una lista de objetos de tipo Logistica que representan el envío.
*/
public static List<Logistica> crearLogistica(double kmCapital, double kmLocalidad, Producto producto,
int cantidad) {
List<Logistica> logisticas = new ArrayList<Logistica>();
double kmTotales = kmCapital + kmLocalidad; double pesoTotal =
cantidad * producto.getTipoProducto().getPeso(); if (kmTotales <=
KMS_MAXIMOS) { while (pesoTotal > PESO_MAXIMO) {
logisticas.add(new LogisticaPropia(PESO_MAXIMO, 0)); pesoTotal -
= PESO_MAXIMO;
}
```

```
if (pesoTotal > 0) { logisticas.add(new
LogisticaPropia(pesoTotal, 0));
}
logisticas.add(new LogisticaPropia(0, kmTotales));
} else { logisticas = calcularLogisticaConCoste(kmCapital, kmLocalidad, producto,
pesoTotal);
}
return logisticas;
}
/**
* Calcula la logística con costo y devuelve una lista de objetos de logística.
*
* @param kmCapital Los kilómetros en ruta hacia la capital.
* @param kmLocalidad Los kilómetros en ruta hacia la localidad.
* @param producto El producto a enviar.
* @param pesoTotal El peso total de los productos.
* @return La lista de objetos de logística.
*/
private static List<Logistica> calcularLogisticaConCoste(double kmCapital, double kmLocalidad,
Producto producto, double pesoTotal) {
List<Logistica> logisticas = new ArrayList<Logistica>();
while (kmLocalidad > KMS_MAXIMOS) {
kmLocalidad -= KMS_MAXIMOS; kmCapital
+= KMS_MAXIMOS;
}
logisticas.add( new PequeñaLogistica(calculaCoste(kmLocalidad, producto, false, pesoTotal), pesoTotal,
kmLocalidad)); if (producto.getTipoProducto().getEsPerecedero()) { logisticas =
calcularLogisticaPorKg(logisticas, kmCapital, producto, pesoTotal);
} else {
```

```
while (kmCapital > KMS_TRAMO) { logisticas = calcularLogisticaPorKg(logisticas,
KMS_TRAMO, producto, pesoTotal); kmCapital -= KMS_TRAMO;
}
if (kmCapital > 0) { logisticas.add(new PequeñaLogistica(calculaCoste(kmCapital, producto, false,
pesoTotal), pesoTotal, kmCapital));
}
}
return logisticas;
}
/**
* Calcula la logística por kilogramo y agrega los objetos de logística a la
* lista existente.
*
* @param logisticas La lista de objetos de logística existente.
* @param kmCapital Los kilómetros en ruta hacia la capital.
* @param producto El producto a enviar.
* @param pesoTotal El peso total de los productos.
* @return La lista de objetos de logística actualizada.
*/
private static List<Logistica> calcularLogisticaPorKg(List<Logistica> logisticas, double kmCapital,
Producto producto, double pesoTotal) { while (pesoTotal > PESO_MAXIMO) { pesoTotal -=
PESO_MAXIMO;
logisticas.add(new GranLogistica(calculaCoste(kmCapital, producto, true, PESO_MAXIMO),
PESO_MAXIMO, 0));
}
if (pesoTotal > 0) { logisticas.add(new GranLogistica(calculaCoste(kmCapital, producto, true,
pesoTotal), pesoTotal, 0));
}
```

```
logisticas.add(new GranLogistica(calculaCoste(kmCapital, producto, true, 0), 0, kmCapital)); return
logisticas;
}
/**
* Calcula el costo de una operación de logística en función de los kilómetros,
* el producto y la cantidad.
*
* @param km Los kilómetros en ruta.
* @param producto El producto a enviar.
* @param granLogistica Indica si es una operación de logística de gran tamaño.
* @param cantidad La cantidad de productos a enviar.
* @return El costo de la operación de logística.
*/
private static double calculaCoste(double km, Producto producto, boolean granLogistica, double
cantidad) { double costo; boolean puedeDescuento = false;
if (!granLogistica) { costo = cantidad * km *
PequeñaLogistica.getPrecioKmKg(); } else { if (cantidad !=
0) { costo = 0.5 * producto.getPrecio(LocalDate.now()) *
cantidad; puedeDescuento = true;
} else { costo = km * 0.05 *
GranLogistica.getPrecioKm();
}
}
if (puedeDescuento) { double descuentoAcumulado = 0.0; for (Entry<Double, Double> entry :
Oferta.OFERTAS_POR_CANTIDAD_DESCUENTO.entrySet()) { double cantidadOferta =
entry.getKey(); double descuento = entry.getValue(); if (cantidad >= cantidadOferta &&
descuentoAcumulado == 0.0) { costo -= costo * descuento; descuentoAcumulado = descuento;
}
}
```

```
}
return costo;
}
}
/**
* Clase que representa una instancia de logística propia.
* @author (Sara Cubero)
* @version (1.0)
*/
public class LogisticaPropia extends Logistica {
/**
* Constructor de la clase LogisticaPropia.
*
* @param kg El peso en kilogramos.
* @param km El recorrido en kilómetros.
*/
public LogisticaPropia(double kg, double km) { super(0,
kg, km);
}
}
/**
* La clase Menus contiene arrays de strings que representan distintos menús para el usuario.
* @author (Sara Cubero)
* @version (1.0)
*/
public class Menus {
/**
* Array de strings que representa el menú principal de la aplicación.
*/
```

```
protected static final String[] MENU_PRINCIPAL = {
"COOPERATIVA EMPRESARIAL",
"¿Qué tipo de gestión desea realizar?",
"1. Productores",
"2. Distribuidores y Consumidores Finales",
"3. Estadísticas",
"0. Salir",
};
/**
* Array de strings que representa el menú de gestión de productores.
*/
protected static final String[] MENU_PRODUCTORES = {
"PRODUCTORES",
"1. Crear Productor",
"2. Imprimir Productores",
"3. Borrar Productor",
"4. Límite en Hectáreas",
"5. Actualizar Precios",
"0. Volver",
};
/**
* Array de strings que representa el menú de creación de productores.
*/
protected static final String[] MENU_CREAR_PRODUCTORES = {
"CREAR PRODUCTOR",
"1. Productor",
"2. Productor federado",
"0. Volver",
};
```

```
/**
* Array de strings que representa el menú de gestión de distribuidores y consumidores finales.
*/
protected static final String[] MENU_COMPRADORES = {
"DISTRIBUIDORES Y CONSUMIDORES FINALES",
"1. Imprimir Distribuidores",
"2. Imprimir Consumidores finales",
"3. Imprimir pedidos",
"4. Crear pedido",
"5. Forzar entregas",
"0. Volver",
};
/**
* Array de strings que representa el menú de gestión de estadística.
*/
protected static final String[] MENU_ESTADISTICA = {
"ESTADÍSTICAS",
"1. Ventas totales",
"2. Importes productores",
"3. Importes logística",
"4. Beneficios",
"5. Evolución precios",
"0. Volver",
};
}
/**
* Clase que representa las ofertas por cantidad de productos con su respectivo
* descuento.
* @author (Sara Cubero)
```

```
* @version (1.0)

*/

import java.util.HashMap;

import java.util.Map; public

class Oferta {

/**

* Mapa que almacena las ofertas por cantidad de productos y su respectivo * descuento. La clave

  representa la cantidad de productos y el valor representa

* el descuento.

*/

public static final Map<Double, Double> OFERTAS_POR_CANTIDAD_DESCUENTO = new
HashMap<>();

static {

// Agregar las ofertas por cantidad y descuento al mapa

OFERTAS_POR_CANTIDAD_DESCUENTO.put(100.0, 0.25);

OFERTAS_POR_CANTIDAD_DESCUENTO.put(50.0, 0.15);

OFERTAS_POR_CANTIDAD_DESCUENTO.put(25.0, 0.5);

}

}

/**

* Clase que representa un pedido realizado por un productor.

* @author (Sara Cubero)

* @version (1.0)

*/

import java.time.LocalDate;

import java.time.LocalDateTime;

import java.util.ArrayList; import

java.util.List; import
```

```
java.util.UUID; public class

Pedido {

/** El plazo máximo en días para la entrega del pedido */ private

static final int PLAZO_MAXIMO_DIAS = 10;

/** El identificador único del pedido */ private

UUID id;

/** La fecha y hora en que se realizó el pedido */

private LocalDateTime fechaPedido; /** La

fecha de entrega del pedido */ private

LocalDate fechaEntrega;

/** Indica si el pedido ha sido entregado */

private boolean entregado;

/** La lista de elementos de pedido */ private

List<ElementoPedido> elementosPedidos;

/** El productor que realizó el pedido */ private

Productor productor;

/**

* Constructor de la clase Pedido.

*

* @param elementosPedidos La lista de elementos de pedido.

* @param fechaEntrega La fecha de entrega del pedido.

*/

public Pedido(List<ElementoPedido> elementosPedidos, LocalDate fechaEntrega, Productor productor) {

this.id = UUID.randomUUID(); this.fechaPedido = LocalDateTime.now(); this.elementosPedidos =

elementosPedidos; this.productor = productor;

this.fechaEntrega = fechaEntrega == null ? (LocalDate.now()).plusDays(PLAZO_MAXIMO_DIAS) :
fechaEntrega;

}

/**
```

```
* Extrae todos los pedidos de todos los compradores.
*
* @return Una lista de todos los pedidos.
*/
public static List<Pedido> extraerPedidos(List<Comprador> compradores) {
List<Pedido> pedidos = new ArrayList<Pedido>(); for (Comprador
comprador : compradores) { pedidos.add(comprador.getPedido());
}
return pedidos;
}
/**
* Obtiene el identificador del pedido.
*
* @return El identificador del pedido.
*/
public UUID getId() { return
id;
}
/**
* Obtiene la fecha y hora en que se realizó el pedido.
*
* @return La fecha y hora del pedido.
*/
public LocalDateTime getFecha() { return
fechaPedido;
}
/**
* Indica si el pedido ha sido entregado.
*
```

```
* @return `true` si el pedido ha sido entregado, `false` en caso contrario.
*/
public boolean getEntregado() { return
entregado;
}
/**
* Establece el estado de entrega del pedido.
*
* @param entregado El estado de entrega del pedido.
*/
public void setEntregado(boolean entregado) { this.entregado
= entregado;
}
/**
* Obtiene el peso total del pedido.
*
* @return El peso total del pedido.
*/
public double getPesoTotal() { double pesoTotal =
0; for (ElementoPedido elemento :
elementosPedidos) { pesoTotal +=
elemento.getPeso();
}
return pesoTotal;
}
/**
* Obtiene la lista de elementos de pedido.
*
* @return La lista de elementos de pedido.
```

```
*/
public List<ElementoPedido> getElementos() { return
elementosPedidos;
}
/**
* Obtiene el productor.
*
* @return el productor.
*/
public Productor getProductor() { return
productor;
}
/**
* Obtiene la fecha de entrega.
*
* @return la fecha de entrega.
*/
public LocalDate getFechaEntrega() { return
fechaEntrega;
}
}
/**
* Clase que representa la logística de tipo pequeña. Esta clase extiende la
* clase abstracta Logistica.
* @author (Sara Cubero)
* @version (1.0)
*/
```

```
public class PequeñaLogistica extends Logistica { /** Precio por
kilómetro por kilogramo para la logística pequeña */ private
static final double PRECIO_KM_KG = 0.01;
/**
* Constructor de la clase PequeñaLogistica.
*
* @param coste El costo de la logística.
* @param kg El peso en kilogramos.
* @param km Los kilómetros recorridos.
*/
public PequeñaLogistica(double coste, double kg, double km) { super(coste,
kg, km);
}
/**
* Obtiene el precio por kilómetro por kilogramo para la logística pequeña.
*
* @return El precio por kilómetro por kilogramo.
*/
public static double getPrecioKmKg() { return
PRECIO_KM_KG;
}
}
/**
* Clase que define los porcentajes para cada tipo de comprador.
* Proporciona constantes estáticas para el porcentaje de distribuidor y el porcentaje de consumidor final.
* @author (Sara Cubero)
* @version (1.0)
*/
public class PorcentajeTipoComprador {
```

```
/**
* Porcentaje para el tipo de comprador Distribuidor.
*/
public static final double PORCENTAJE_DISTRIBUIDOR = 5;
/**
* Porcentaje para el tipo de comprador Consumidor Final.
*/
public static final double PORCENTAJE_CONSUMIDOR_FINAL = 15;
}
/**
* Clase que se encarga de generar y actualizar precios únicos para los
* productos.
* @author (Sara Cubero)
* @version (1.0)
*/
import java.time.LocalDate;
import java.util.List; import
java.util.Random; public class
PrecioGenerator {
/**
* Actualiza los precios únicos de los productos en la tabla de precios.
*
* @param productos La lista de productos.
*/
public static void actualizarPreciosUnicos(List<Producto> productos) { for
(TipoProducto tipoProducto : TipoProducto.values()) {
Producto producto = buscarProductoPorTipo(productos, tipoProducto); if
(producto != null) {
TablaPrecios.setPrecio(numeroRandom(), producto.getTipoProducto(), generarFechaAleatoria());
```

```
}

}

}

/**

* Busca un producto en la lista de productos por su tipo.

*

* @param productos La lista de productos.

* @param tipoProducto El tipo de producto a buscar.

* @return El producto encontrado o `null` si no se encuentra.

*/

private static Producto buscarProductoPorTipo(List<Producto> productos, TipoProducto tipoProducto) {

for (Producto producto : productos) { if (producto.getTipoProducto() == tipoProducto) { return producto;

}

}

return null;

}

/**

* Genera un número aleatorio entre 0.1 y 1 (con dos decimales).

*

* @return El número aleatorio generado.

*/

private static double numeroRandom() { return

Math.round((Math.random() * (1 - 0.1) + 0.1) * 100) / 100.0;

}

/**

* Genera una fecha aleatoria dentro del año actual.

*

* @return La fecha aleatoria generada.

*/
```

```
public static LocalDate generarFechaAleatoria() { Random random =

new Random(); int minDay = 1; int maxDay =

LocalDate.now().getDayOfYear(); int randomDay =

random.nextInt(maxDay - minDay + 1) + minDay; return

LocalDate.ofYearDay(LocalDate.now().getYear(), randomDay);

}

}

/**

* Clase que representa un producto y sus atributos.

* @author (Sara Cubero)

* @version (1.0)

*/

import java.time.LocalDate;

import java.util.ArrayList;

import java.util.List; public

class Producto {

/* Nombres de los atributos para imprimir */ private static final String[]

NOMBRES_ATRIBUTOS = { "hectareas", "cantidad", "pesoTotal" };

/* Iva estándar */ private static final

double IVA = 0.10; private

TipoProducto tipoProducto; private

double hectareas;

/**

* Constructor de la clase Producto.

*

* @param tipoProducto El tipo de producto.

* @param hectareas Las hectáreas del producto.

*/
```

```
public Producto(TipoProducto tipoProducto, double hectareas) {
this.tipoProducto = tipoProducto; this.hectareas = hectareas;
}
/**
* Crea una lista de productos a partir de las hectáreas totales disponibles.
*
* @param hectareasTotales Las hectáreas totales disponibles.
* @return La lista de productos creados.
*/
public static List<Producto> crearProductos(double hectareasTotales) {
List<Producto> productos = new ArrayList<Producto>(); boolean opcion;
do {
System.out.println("Datos de producto:");
TipoProducto[] tiposProducto = TipoProducto.values(); for
(int i = 0; i < tiposProducto.length; i++) {
System.out.println((i + 1) + "- " + tiposProducto[i].getNombre());
}
int indiceTipoProducto = Utilidades.getInt("Ingrese el número correspondiente al tipo de producto: ") - 1;
TipoProducto tipoProducto = tiposProducto[indiceTipoProducto]; double hectareas =
validarHectareas(hectareasTotales); hectareasTotales -= hectareas;
Producto producto = new Producto(tipoProducto, hectareas);
productos.add(producto); opcion =
Utilidades.getBoolean("Agregar otro producto:");
System.out.println(); }
while (opcion);
return productos;
}
/**
* Valida el número de hectáreas ingresado por el usuario.
```

```
*
* @param hectareasTotales Las hectáreas totales disponibles.
* @return Las hectáreas válidas ingresadas por el usuario.
*/
public static double validarHectareas(double hectareasTotales) {
boolean valido = false; double hectareas; do {
hectareas = Utilidades.getDouble("Hectareas: ", 0.0);
if (hectareas <= hectareasTotales) { return
hectareas;
}
System.out.println("Las hectareas de producto no pueden ser más de las totales");
} while (!valido); return
hectareas;
}
/**
* Imprime todos los productos de la lista.
*
* @param productos La lista de productos a imprimir.
*/
public static void imprimirTodos(List<Producto> productos) { for
(Producto producto : productos) { imprimirElemento(producto);
}
}
/**
* Imprime los atributos de un producto.
*
* @param producto El producto a imprimir.
*/
public static void imprimirElemento(Producto producto) {
```

```
Utilidades.imprimirObjeto(producto.tipoProducto, TipoProducto.NOMBRES_ATRIBUTOS);

Utilidades.imprimirObjeto(producto, NOMBRES_ATRIBUTOS);

}

/**

* Actualiza el precio de un producto en la tabla de precios.

*

* @param producto El producto cuyo precio se actualizará.

* @param precio El nuevo precio del producto.

*/

public static void actualizarPrecio(Producto producto, double precio) {

TablaPrecios.setPrecio(precio, producto.getTipoProducto(), LocalDate.now());

System.out.println("El precio del producto " + producto.getTipoProducto().getNombre()

+ " ha sido actualizado a " + precio);

}

/**

* Extrae todos los productos de todos los productores.

*

* @return Una lista de todos los productos.

*/

public static List<Producto> extraerProductos(List<Productor> productores) {

List<Producto> productos = new ArrayList<Producto>(); for (Productor

productor : productores) { productos.addAll(productor.getProductos());

}

return productos;

}

/**

* Devuelve el tipo de producto.

*

* @return El tipo de producto.
```

```
*/
public TipoProducto getTipoProducto() { return
tipoProducto;
}
/**
* Devuelve el precio del producto para una fecha especificada.
*
* @param fecha La fecha para la cual se obtendrá el precio.
* @return El precio del producto.
*/
public double getPrecio(LocalDate fecha) { return
TablaPrecios.getPrecio(tipoProducto, fecha);
}
/**
* Devuelve el precio del producto con el IVA incluido para una fecha
* especificada.
*
* @param fecha La fecha para la cual se obtendrá el precio con IVA.
* @return El precio del producto con IVA.
*/
public double getPrecioIVA(LocalDate fecha) { return
(getPrecio(fecha) * IVA) + getPrecio(fecha);
}
/**
* Devuelve la cantidad de productos basándose en las hectáreas y el rendimiento
* por hectárea.
*
* @return La cantidad de productos.
*/
```

```
public int getCantidad() {

return (int) Math.round(tipoProducto.getRendimientoHectarea() * hectareas);

}

/**

* Devuelve el peso total de los productos.

*

* @return El peso total de los productos.

*/

public double getPesoTotal() { return

Math.round(tipoProducto.getPeso() * getCantidad());

}

/**

* Devuelve las hectáreas del producto.

*

* @return Las hectáreas del producto.

*/

public double getHectareas() { return

hectareas;

}

}

import java.util.ArrayList; import java.util.List; import java.util.UUID; public abstract

class Productor { /*Constante para imprmir*/ private static final String[]

NOMBRES_ATRIBUTOS = { "id", "nombre", "hectareas" }; private UUID id; private

String nombre; private double hectareas; private List<Producto> productos;


/**

* Constructor de la clase Productor.

*

* @param nombre
```

```
* @param hectareas * @param productos

*/

public Productor(String nombre, double hectareas, List<Producto> productos) {

this.id = UUID.randomUUID(); this.nombre = nombre; this.hectareas =

hectareas; this.productos = productos;

}

/**

*

*/

public static Productor crearProductor() {

String nombre = Utilidades.getString("Nombre de productor: "); double hectareas

= Utilidades.getDouble("Hectareas totales de productor: ", 0.0);

Productor productor = ProductorFactory.crearProductor(nombre, hectareas,
Producto.crearProductos(hectareas)); imprimirElemento(productor);

Producto.imprimirTodos(productor.productos); return

productor;

}

/**

* Crea un productor federado a partir de una lista de productores, un nombre de producto,

* una lista de nombres de productores y un indicador de aleatoriedad.

*

* @param productores la lista de productores disponibles.

* @param producto el nombre del producto deseado.

* @param productoresNombres la lista de nombres de productores.

* @param random un indicador de aleatoriedad.

* @return el objeto ProductorFederado creado.

* @throws IllegalArgumentException si los productores o el producto proporcionados no son correctos.

*/

public static ProductorFederado crearProductorFederado(List<Productor> productores, String producto,
List<String> productoresNombres, boolean random) {
```

```
List<ProductorPequeño> productoresPequeños = new ArrayList<ProductorPequeño>();

List<ProductorFederado> productoresFederados = new ArrayList<ProductorFederado>();

Producto productoElegido = null; for (Productor productorItem : productores) { if

(productorItem instanceof ProductorPequeño) { if

(productoresNombres.contains(productorItem.getNombre())) { for (Producto productoItem

: productorItem.productos) { if

(productoItem.getTipoProducto().getNombre().equals(producto)) {

productoresPequeños.add((ProductorPequeño) productorItem); productoElegido =

productoItem;

}

}

} }

else if (productorItem instanceof ProductorFederado) {

productoresFederados.add((ProductorFederado) productorItem);

}

}

if (productoElegido == null || productoresPequeños.size() == 0) { throw new

IllegalArgumentException("Los productores o producto aportados no son correctos"); }


return ProductorFederado.setProductorFederado(productoresPequeños, productoresFederados,
productoElegido, random);

}

/**

* Imprime los detalles de todos los productores en la lista dada.

*

* @param listaProductores la lista de productores a imprimir.

*/

public static void imprimirTodos(List<Productor> listaProductores) { for

(Productor productor : listaProductores) { imprimirElemento(productor);

System.out.println("Tipo de productor: "+ tipoProductor(productor)+ "\n");
```

```
System.out.println("Sus productos:");

Producto.imprimirTodos(productor.productos);

System.out.println("");

}

}

/**

* Imprime los detalles del productor dado.

*

* @param productor el objeto Productor a imprimir.

*/

public static void imprimirElemento(Productor productor) {

Utilidades.imprimirObjeto(productor, NOMBRES_ATRIBUTOS);

}

/**

* Borra un productor de la lista de productores.

*

* @param productores la lista de productores.

* @return la lista de productores actualizada después de eliminar el productor.

*/

public static List<Productor> borrarProductor(List<Productor> productores) { for

(int i = 0; i < productores.size(); i++) {

System.out.println((i + 1) + "- " + productores.get(i).getNombre());

}

int indice = Utilidades.getInt("¿Qué productor desea borrar? (Ingrese el número)") - 1; if

(indice >= 0 && indice < productores.size()) {

Productor productorEliminado = productores.remove(indice);

System.out.println("Productor " + productorEliminado.getNombre() + " eliminado correctamente."); }

else {

System.out.println("No se encontró el productor.");
```

```
}

return productores;

}

/**

* Devuelve el tipo de productor del productor dado.

*

* @param productor el objeto Productor para el cual se desea obtener el tipo.

* @return el tipo de productor.

*/

public static String tipoProductor(Productor productor) {

if (productor instanceof ProductorGrande) { return

TipoProductor.GRANDE;

} else if (productor instanceof ProductorPequeño) { return

TipoProductor.PEQUEÑO;

}

return TipoProductor.FEDERADO;

}

/**

* Obtiene el número total de hectáreas del productor.

*

* @return el número total de hectáreas del productor.

*/

public String getNombre() { return

nombre;

}

public double getHectareas() { return

hectareas;

}

/**
```

```
* Obtiene el ID del productor.
*
* @return el ID del productor.
*/
public UUID getId() { return
id;
}
/**
* Obtiene la lista de productos del productor.
*
* @return la lista de productos del productor.
*/
public List<Producto> getProductos() { return
productos;
}
/**
* Establece el nombre del productor.
*
* @param nombre el nuevo nombre del productor.
*/
public void setNombre(String nombre) { this.nombre
= nombre;
}
}
/**
* Clase que se encarga de crear objetos de tipo Productor según los parámetros
* proporcionados.
* @author (Sara Cubero)
* @version (1.0)
```

```
*/
import java.util.List; public
class ProductorFactory {
/**
* Crea un objeto Productor según los parámetros especificados.
*
* @param nombre El nombre del productor.
* @param hectareas Las hectáreas del productor.
* @param productos Los productos del productor.
* @return El objeto Productor creado.
* @throws IllegalArgumentException Si se intenta crear un ProductorPequeño con
* más de 5 productos.
*/
public static Productor crearProductor(String nombre, double hectareas, List<Producto> productos) {
LimiteHectareas limite = LimiteHectareas.getInstance(); double limiteHectareas = limite.getLimite();

if (hectareas > limiteHectareas) { return new
ProductorGrande(nombre, hectareas, productos);
} else { if (productos.size() >
ProductorPequeño.getMaxProductos()) {
throw new IllegalArgumentException("No se puede crear un ProductorPequeño con más de 5 productos");
}
return new ProductorPequeño(nombre, hectareas, productos);
}
}
}
/**
* Clase que representa a un Productor Federado, que es una agrupación de
* Productores Pequeños.
```

```
* @author (Sara Cubero)

* @version (1.0)

*/

import java.util.ArrayList; import java.util.List; public

class ProductorFederado extends Productor {

/**

* Crea un objeto ProductorFederado con el nombre, hectáreas y productos

* especificados.

*

* @param nombre El nombre del productor federado.

* @param hectareas Las hectáreas totales del productor federado.

* @param productos Los productos del productor federado.

*/

public ProductorFederado(String nombre, double hectareas, List<Producto> productos) {
super(nombre, hectareas, productos);

}

/**

* Crea un objeto ProductorFederado a partir de una lista de Productores

* Pequeños y un Producto. Los Productores Pequeños se agrupan en un Productor * Federado, sumando

  sus hectáreas y creando un nuevo Producto con las hectáreas

* totales.

*

* @param productores La lista de Productores Pequeños.

* @param productoresFederados La lista de Productores Federados existentes.

* @param producto El Producto que se utilizará para crear el nuevo

* Productor Federado.

* @return El objeto ProductorFederado creado.

* @throws IllegalArgumentException Si la suma de las hectáreas de los

* Productores Pequeños supera el límite de un
```

```
* Productor Pequeño.

*/

public static ProductorFederado setProductorFederado(List<ProductorPequeño> productores,

List<ProductorFederado> productoresFederados, Producto producto, boolean random) {

LimiteHectareas limite = LimiteHectareas.getInstance(); double limiteHectareas =

limite.getLimite(); double totalHectareas = 0; if (totalHectareas > limiteHectareas) {

throw new IllegalArgumentException("No se puede crear el productor federado, "

+ "ya que la suma de las hectareas supera el límite de productor pequeño");

}

String nombreFederado = ""; double

hectareas = 1.0; for (Productor

productor : productores) { do {


if (!random) {

System.out.println("Hectareas totales del productor:" + productor.getHectareas()); hectareas

= Utilidades.getDouble("Cantidad de hectareas de " + productor.getNombre(), 0.0);

}

} while (hectareas > productor.getHectareas()); if

(nombreFederado.isEmpty()) { nombreFederado =

productor.getNombre() + " (" + hectareas + ")";

} else { nombreFederado += " - " + productor.getNombre() + " (" +

hectareas + ")";

}

totalHectareas += hectareas;

}

List<Producto> productos = new ArrayList<Producto>();

Producto productoFederado = new Producto(producto.getTipoProducto(), totalHectareas);

productos.add(productoFederado); return new ProductorFederado(nombreFederado,

totalHectareas, productos);
```

```
}

}

/**

* Clase que se encarga de generar Productores aleatorios.

* @author (Sara Cubero)

* @version (1.0)

*/

import java.util.ArrayList; import

java.util.Arrays; import

java.util.List; import

java.util.Random; public class

ProductorGenerator { private

static final List<TipoProducto>

PRODUCTOS =

Arrays.asList(TipoProducto.values

()); private static final Random

RAND = new Random();

/**

* Genera un Productor aleatorio.

*

* @param isSmall Indica si se debe generar un Productor pequeño.

* @return El objeto Productor generado.

*/

public static Productor generarProductorRandom(boolean isSmall) {

String nombre = generarNombreRandom(); double hectareas =

isSmall ? 2 : generarHectareasRandom();

List<Producto> productos = new ArrayList<>();

TipoProducto producto = PRODUCTOS.get(RAND.nextInt(PRODUCTOS.size()));
```

```
double cantidad = isSmall ? 2 : Math.max(0.1, Math.min(Math.round(RAND.nextDouble() * 10),
hectareas)); productos.add(new Producto(producto, cantidad)); return
ProductorFactory.crearProductor(nombre, hectareas, productos);
}
/**
* Genera un nombre aleatorio para un Productor.
*
* @return El nombre generado.
*/
private static String generarNombreRandom() {
String[] nombres = { "Juan", "Pedro", "Luis", "Ana", "María", "Lucía", "Miguel", "José", "Carla", "Elena",
"Pablo", "Alberto" };
String[] apellidos = { "García", "Rodríguez", "Martínez", "López", "González", "Pérez", "Sánchez",
"Ramírez",
"Torres", "Flores", "Vargas", "Castillo" }; int indexNombre =
new Random().nextInt(nombres.length); int indexApellido =
new Random().nextInt(apellidos.length); return
nombres[indexNombre] + " " + apellidos[indexApellido];
}
/**
* Genera un número aleatorio de hectáreas para un Productor.
*
* @return El número de hectáreas generado.
*/
private static double generarHectareasRandom() {
double hectareas = 0; while (hectareas == 0) {
hectareas = Math.round(RAND.nextDouble() * 20);
}
return hectareas;
}
```

```
}

/**

* Clase que representa un Productor grande.

* Extiende la clase abstracta Productor.

* @author (Sara Cubero)

* @version (1.0)

*/

import java.util.List; public class

ProductorGrande extends Productor

{

/**

* Constructor de la clase ProductorGrande.

*

* @param nombre El nombre del Productor.

* @param hectareas El número de hectáreas del Productor.

* @param productos La lista de productos del Productor.

*/

public ProductorGrande(String nombre, double hectareas, List<Producto> productos) {

super(nombre, hectareas,productos);

}

}

/**

* Clase que representa un Productor pequeño. Extiende la clase abstracta Productor.

* @author (Sara Cubero)

* @version (1.0)

*/

import java.util.List; public class ProductorPequeño

extends Productor {

/**
```

```
* Constante que define el número máximo de productos permitidos para un
* Productor pequeño.
*/
private static final int MAX_PRODUCTOS = 5;
/**
* Constructor de la clase ProductorPequeño.
*
* @param nombre El nombre del Productor.
* @param hectareas El número de hectáreas del Productor.
* @param productos La lista de productos del Productor.
*/
public ProductorPequeño(String nombre, double hectareas, List<Producto> productos) {
super(nombre, hectareas, productos);
}
/**
* Obtiene el número máximo de productos permitidos para un Productor pequeño.
*
* @return El número máximo de productos permitidos.
*/
public static int getMaxProductos() {
return MAX_PRODUCTOS;
}
}
/**
* Clase para establecer y actualizar precios derivados de productos
* @author (Sara Cubero)
* @version (1.0)
*/
```

```
import java.time.LocalDate;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List; import

java.util.Map; public class

TablaPrecios {

// Listas para almacenar los precios actuales.

private static final Map<TipoProducto, Double> precios = new HashMap<>();

// Listas para almacenar los precios anteriores.

private static final List<Map<LocalDate, Map<TipoProducto, Double>>> preciosAnteriores = new
ArrayList<>(); static {

precios.put(TipoProducto.ACEITE, 3.0);

precios.put(TipoProducto.ACEITUNAS, 2.5);

precios.put(TipoProducto.ACELGA, 1.0);

precios.put(TipoProducto.ALGODON, 1.8);

precios.put(TipoProducto.ARROZ, 0.60);

precios.put(TipoProducto.BROCOLI, 2.0);

precios.put(TipoProducto.CEBOLLA, 0.40);

precios.put(TipoProducto.CIRUELOS, 0.80);

precios.put(TipoProducto.COLIFLOR, 2.0);

precios.put(TipoProducto.ESPINACAS,

1.80); precios.put(TipoProducto.LECHUGA,

0.60); precios.put(TipoProducto.MAIZ, 0.80);

precios.put(TipoProducto.MANZANA, 0.65);

precios.put(TipoProducto.MELOCOTONER

OS, 0.90);

precios.put(TipoProducto.NARANJOS,

0.68); precios.put(TipoProducto.TOMATE,

0.85); precios.put(TipoProducto.TRIGO,
```

```
0.40);
precios.put(TipoProducto.ZANAHORIA,
0.10); precios.put(TipoProducto.PERA,
0.60);
}
/**
* Obtiene si el precio del producto que se ajusta mas a la fecha requerida.
* @return precio
*/
public static double getPrecio(TipoProducto tipoProducto, LocalDate fecha) { double
precio = precios.getOrDefault(tipoProducto, 0.0); for (Map<LocalDate,
Map<TipoProducto, Double>> registroPrecios : preciosAnteriores) { if
(registroPrecios.containsKey(fecha)) {
Map<TipoProducto, Double> preciosFecha = registroPrecios.get(fecha);
if (preciosFecha.containsKey(tipoProducto)) { precio =
preciosFecha.get(tipoProducto); break;

}
}
}
return precio;
}
/**
* Actualiza el precio de un tipo de producto
*
*/
public static void setPrecio(double precio, TipoProducto tipoProducto, LocalDate fecha) {
Map<TipoProducto, Double> preciosAnterioresMap = new HashMap<>(precios);
Map<LocalDate, Map<TipoProducto, Double>> registroPrecios = new HashMap<>();
```

```
registroPrecios.put(fecha, preciosAnterioresMap);

preciosAnteriores.add(registroPrecios); precios.put(tipoProducto, precio);

}

public static List<Map<LocalDate, Map<TipoProducto, Double>>> getPreciosAnteriores(TipoProducto
tipoProducto) { return preciosAnteriores;

}

/**

* Obtiene si el precio del producto a sufrido cambios.

*

* @return boolean

*/

public static boolean productoTieneCambios(TipoProducto tipoProducto) {

List<Map<LocalDate, Map<TipoProducto, Double>>> preciosAnteriores = TablaPrecios

.getPreciosAnteriores(tipoProducto); if

(preciosAnteriores.isEmpty()) { return

false;


}

Map<LocalDate, Map<TipoProducto, Double>> preciosMap =

preciosAnteriores.get(preciosAnteriores.size()  -  1);  if

(!preciosMap.containsKey(LocalDate.now())) { return false;

}

double precioActual = TablaPrecios.getPrecio(tipoProducto, LocalDate.now());

double precioAnterior = preciosMap.get(LocalDate.now()).get(tipoProducto); return

precioActual != precioAnterior;

}

}

/**

* Clase que define los tipos de compradores disponibles.

* Proporciona constantes estáticas para los tipos de comprador "Distribuidor" y "Consumidor final".
```

```
* @author (Sara Cubero)

* @version (1.0)

*/

public class TipoComprador {

/**

* Tipo de comprador "Distribuidor".

*/

public final static String DISTRIBUIDOR = "Distribuidor";

/**

* Tipo de comprador "Consumidor final".

*/

public final static String CONSUMIDOR_FINAL = "Consumidor final";

}

/**

* Clase para establecer productos

* @author (Sara Cubero)

* @version (1.0)

*/

public enum TipoProducto {

ACEITE(1, "Aceite de oliva", false, 1000, 0.75),

ACEITUNAS(2, "Aceitunas", true, 2000, 0.2),

ACELGA(3, "Acelga", true, 2000, 0.3),

ALGODON(4, "Algodón", false, 3000, 0.05),

ARROZ(5, "Arroz", false, 1000, 0.1),

BROCOLI(6, "Brócoli", true, 1500, 0.5),

CEBOLLA(7, "Cebolla", true, 5000, 0.2),

CIRUELOS(8, "Ciruelos", true, 2000, 0.1),

COLIFLOR(9, "Coliflor", true, 4400, 0.7),

ESPINACAS(10, "Espinacas", true, 3690, 0.2),
```

```
LECHUGA(11, "Lechuga", true, 1200, 0.2),

MAIZ(12, "Maíz", false, 9000, 0.15),

MANZANA(13, "Manzana", true, 3300, 0.2),

MELOCOTONEROS(14, "Melocotoneros", true, 1700, 0.1),

NARANJOS(15, "Naranjos", true, 1800, 0.2),

TOMATE(16, "Tomate", true, 2500, 0.1),

TRIGO(17, "Trigo", false, 4000, 0.05),

ZANAHORIA(18, "Zanahoria", true, 2000, 0.1),

PERA(19, "Pera", true, 2200, 0.2); /*Constante

para imprimir*/

protected static final String[] NOMBRES_ATRIBUTOS = { "id", "nombre", "esPerecedero",
"rendimientoHectarea", "peso" };

private final int id; private final

String nombre; private final

boolean esPerecedero; private

final double

rendimientoHectarea; private

final double peso;

/**

* Constructor de la clase TipoProducto.

*

* @param id

* @param nombre

* @param esPerecedero

* @param rendimientoHectarea

* @param peso

*/

TipoProducto(int id, String nombre, boolean esPerecedero, double rendimientoHectarea, double peso) {

this.id = id;
```

```
this.nombre = nombre; this.esPerecedero =
esPerecedero; this.rendimientoHectarea =
rendimientoHectarea; this.peso = peso;
}
/**
* Devuelve el id del tipo de producto
*
* @return id
*/
public int getId() { return
id;
}
/**
* Devuelve el nombre del tipo de producto
*
* @return nombre
*/
public String getNombre() { return
nombre;
}
/**
* Devuelve si es perecedero el tipo de producto
*
* @return esPerecedero
*/
public boolean getEsPerecedero() { return
esPerecedero;
}
/**
```

```
* Devuelve el rendimiento por hectarea de un tipo de producto.
*
* @return rendimientoHectarea
*/
public double getRendimientoHectarea() { return
rendimientoHectarea;
}
/**
* Devuelve el peso de un producto.
*
* @return peso.
*/
public double getPeso() {
return peso;

}
}
/**
* Clase que define los tipos de productor disponibles.
* @author (Sara Cubero)
* @version (1.0)
*/
public class TipoProductor {
/**
* Constante que representa el tipo de productor "grande".
*/
public final static String GRANDE = "grande";
/**
* Constante que representa el tipo de productor "pequeño".
```

```
*/
public final static String PEQUEÑO = "pequeño";
/**
* Constante que representa el tipo de productor "federado".
*/
public final static String FEDERADO = "federado";
}
/**
* Clase de utilidades que proporciona métodos útiles para la interacción con el
* usuario y la manipulación de objetos.
* @author (Sara Cubero)
* @version (1.0)
*/
import java.io.BufferedReader; import
java.io.IOException; import
java.io.InputStreamReader; import
java.lang.reflect.Method; public class
Utilidades {
/**
* Imprime un menú en la consola con las opciones proporcionadas.
*
* @param opciones Las opciones del menú.
*/
public static void imprimirMenu(String[] opciones) { for
(String opcion : opciones) {
System.out.println(opcion);
}
}
/**
```

```
* Imprime los atributos de un objeto utilizando reflexión.
*
* @param objeto El objeto del cual imprimir los atributos.
* @param atributos Los nombres de los atributos a imprimir.
*/
public static void imprimirObjeto(Object objeto, String[] atributos) {
StringBuilder sb = new StringBuilder(); for (String atributo :
atributos) { try {
Method metodo = objeto.getClass()
.getMethod("get" + atributo.substring(0, 1).toUpperCase() + atributo.substring(1));
Object valor = metodo.invoke(objeto); sb.append(atributo).append(":
").append(valor).append("\n"); } catch (Exception e) {
e.printStackTrace();
}
}
System.out.println(sb.toString());
}
/**
* Lee un número entero desde la entrada estándar.
*
* @param mensaje El mensaje que se muestra al usuario.
* @return El número entero ingresado.
*/
public static int getInt(String mensaje) {
BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
boolean entradaValida = false; int numero = 0; while (!entradaValida) { try {
System.out.print(mensaje); numero =
Integer.parseInt(reader.readLine());
entradaValida = true;
```

```
} catch (NumberFormatException e) {
System.out.println("Error: debes ingresar un número entero.");
} catch (IOException e) {
System.out.println("Error al leer la entrada del usuario.");
}
}
return numero;
}
/**
* Lee un número decimal desde la entrada estándar.
*
* @param mensaje El mensaje que se muestra al usuario.
* @param limit El límite máximo para el número decimal. 0.0 si no hay límite.
* @return El número decimal ingresado.
*/
public static double getDouble(String mensaje, double limit) {
BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
boolean entradaValida = false; double numero = 0; while (!entradaValida) { try {
System.out.print(mensaje); numero =
Double.parseDouble(reader.readLine()); if (limit
== 0.0 || numero <= limit) entradaValida = true;
else
System.out.println("Error: la cantidad ingresada no puede ser superior a: " + limit + ".");
} catch (NumberFormatException e) {
System.out.println("Error: debes ingresar un número decimal.");
} catch (IOException e) {
System.out.println("Error al leer la entrada del usuario.");
}
}
```

```
return numero;

}

/**

* Lee una cadena de texto desde la entrada estándar.

*

* @param mensaje El mensaje que se muestra al usuario.

* @return La cadena de texto ingresada.

*/

public static String getString(String mensaje) {

BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

String input = ""; try {

System.out.print(mensaje); input

= reader.readLine();

} catch (IOException e) {

System.out.println("Error al leer la entrada del usuario.");

}

return input;

}

/**

* Lee una respuesta booleana ("S" o "N") desde la entrada estándar.

*

* @param mensaje El mensaje que se muestra al usuario.

* @return `true` si la respuesta es "S", `false` si la respuesta es "N".

*/

public static boolean getBoolean(String mensaje) {

BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

boolean input = false; boolean valido = false; while (!valido) { try {

System.out.print(mensaje + " (S/N): ");
```

```
String respuesta = reader.readLine().toUpperCase();

if (respuesta.equals("S")) { input = true;


valido = true;

} else if (respuesta.equals("N")) {

input = false; valido = true;

} else {

System.out.println("La respuesta ingresada no es válida. Por favor, ingrese S o N.");

}

} catch (IOException e) {

System.out.println("Error al leer la entrada del usuario.");

}

}

return input;

}

}
```