PRÁCTICA OBLIGATORIA

Tomás Solano Campos

PRÁCTICA OBLIGATORIA PARA EL CURSO 2022/2023:

Tomas Solano

Índice:

1. Objetivo

Me pongo en el supuesto en el que hay una corporativa con una horrible organización y me dispongo a proponer una solución a la corporativa de realizar una corporativa mucho más eficiente de la que existe, con el fin de ahorrar costos y proponer un sistema que permite una comunicación directa entre corporativa ,logística y productores.

Para realizar esto he realizado un programa en BlueJ con Java.

Gracias a este programa se mejorará la eficiencia de la corporativa y se ahorrará mucho dinero.

2. Especificaciones

2.1 Especificaciones funcionales

I. Permite crear y modificar tantos productos ,productores, empresas…
II. Puedes crear listas generales tanto de las mencionadas antes ,además de manipular la información que te llega.
III. Permite realizar pedidos y mostrarlos. Así como calcular el costo de la logística implicada , calcular impuestos y demás.
IV. Permite crear y personalizar tus propias empresas y la forma en la que guardas cada uno de tus objetos
V. Utiliza una jerarquía que puede ser tan genérica o especifica como tú quieras. Puesto que utiliza en su mayoría métodos generales para formar los objetos que son heredados y guardados en listas en las que puedes guardar lo que quieras. Por ejemplo ,puedes tener tu propia lista de productores federados o meterlo a la fuerza con todos los demás productores.

2.2 Especificaciones no funcionales

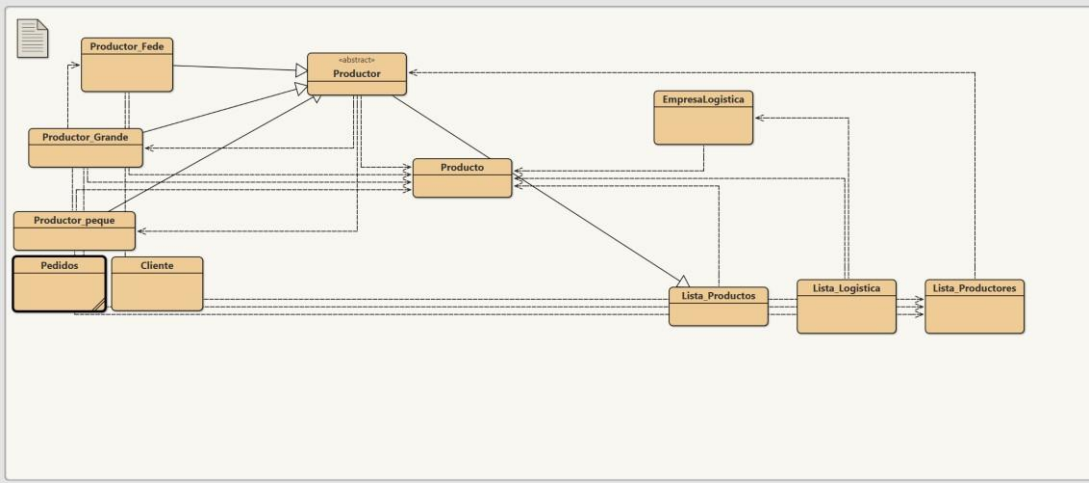
I. No pedían en ningún momento en la práctica un menú así que no lo he hecho, resultando en que cada prueba de funcionamiento tuviera que declarar todo manualmente. Por ejemplo, para declarar un productor tienes que hacer un producto ,guardarlo en su lista y declarar un productor
II. He hecho que los precios puedan ser elegidos por ti sin ninguna clase de restricción, pudiendo poner precios desorbitados. Esto debido a que asumo que quien va a usar la aplicación es un trabajador de la empresa que sabe cómo funciona el programa.
III. No hay fechas. Significando que no hay una lista de ventas al final del año. IV. No tengo un Main.

3. Modelado de objetos

El programa ha sido diseñado pensando en un modelo dinámico . Al principio probé a hacer un modelo más estático en los cuales se usaban hashmaps para coger las listas :

```
protected HashMap<Producto,Double> listaProductores;
//Guarda una lista de productos (cultivados por el productor) con su extensión
//<NombreProducto, Extension>
protected HashMap<Integer,HashMap<Producto,Double>> importeProductor;

//<Producto, Importe>>

/**
 * Constructor clase Productor
 *
 */
public Productor(String n)
{
    nombre=n;
    extensionTotal = 0;
    listaProductores = new HashMap<>();
    listaProductores = new HashMap<>();
}

/**
 * Constructor clase Productor(Productor con productos)
 *
 *
 */
public Productor(String n, HashMap<Producto,Double> p)
{
    //Para crear un productor con los productos ya añadidos, le podemos pasar un HashMap<Producto,Extension> que podriamos crear en AlmacenProdu
    nombre=n;
    extensionTotal = 0;
    listaProductores = p;
    //Tomamos todos los pares producto-extension y sumamos las extensiones
    for (HashMap.Entry<Producto, Double> entrada : p.entrySet()){

        extensionTotal += entrada.getValue();
         listaProductores = new HashMap<>();
        }
}
```

pero me di cuenta de que tenía muchos problema almacenando información y lo descarté por el modelo dinámico aprovechando la interfaz de BlueJ me fue muy fácil crear métodos polimorfismos que utilizaban sus listas propias ,es decir , tú puedes seleccionar en que lista puedes guardar a tu productor ,producto... Etc. ,esto fue una gran ventaja a la hora de programar a los productores federados puesto que puedes hacerte una lista única para ellos y mantener un mejor orden ,a su vez como la lista es general se hacen distinciones a la hora de sacar su producto con un instanceof. Este método también lo he aplicado a los productos ,creando constructores especiales para guardar en una lista u otra .Quedando algo tal que así:



Los dos únicos problemas que tienen está perspectiva es que el tiempo de pruebas es muy lento ,tienes que asegurarte de crear los objetos que necesitas con sus parámetros exactos para testear un método que ocupe esos objetos .

El segundo problema fue el cambio de perspectiva en mi mente ,el programa ya no iba dirigido a un público general ,sino a un trabajador que organizaba listas , productos y gestionaba los distintos pedidos que le llegaban . Con esta perspectiva asumí que el cliente sabía qué hacer en cada momento y omití cierta generación de excepción que me parecía

innecesaria. Por último, con el trabajador en mente, decidí crear un método para cambiar el precio del producto, en donde el supuesto trabajador recibiría una tabla de cambios de precios y cambiaría uno a uno los precios. Esto le consumiría mucho tiempo , así que estuve probando una clase para cambiar fechas:

,que no llego a la luz dado a que rompía la percepción que tenía de mi programa y a que no me daba tiempo a implementarla correctamente ,por ende, decidí borrarla completamente.

```java
import java.time.*;

public class obtenerFecha
{
    private LocalDate fecha;
    public obtenerFecha() {
        fecha = LocalDate.now();
    }
       public obtenerFecha(LocalDate date) {
        fecha = date;
    }
    public int getDay() {
        return fecha.getDayOfMonth();
    }
    public int getMonth() {
        return fecha.getMonthValue();
    }
    public int getYear() {
        return fecha.getYear();
    }
    public obtenerFecha(int day, int month, int year) {
        try {
            fecha = LocalDate.of(year, month, day);
        } catch(DateTimeException e) {
            System.out.println("Invalid date, please try again.");
        }
    }
    public void setDate(int day, int month, int year) {
        fecha = LocalDate.of(year, month, day);
    }
    public Fecha addDays(int days) {
        return new Fecha(fecha.plusDays(days));
    }
    public int getWeekOfTheYear() {
        int week = fecha.getDayOfYear() / 7;
        int weekDay = fecha.getDayOfYear() % 7;
        int firstDay = fecha.ofYearDay(fecha.getYear(), 1).getDayOfWeek().ordinal();
        if(weekDay + firstDay >= 7){
            week++;
        }
        return week;
    }
```

Una posible utilización de esta clase hubiera sido la de cambiar el precio cada semana, pero acabé descartándolo como dije antes.

```java
     *
     * @precon  Tiene que haber pasado una semana
     */
    public void cambiarPreciosProducutos(){
        for (Producto producto : productos) {
            producto.cambiarPrecio((Math.round(Math.random()*500+1))/100); //los precios estaran entre 0.1 y 5€/kg
        }
    }
```

Por último, en cuanto al diseño de clases, he creado relaciones de dependencia entre padres e hijos, así como adaptados métodos a la herencia de su padre y a alguna característica del hijo o directamente creados métodos polimórficos ,ejemplos claros de estos se pueden ver en los productos. De los que hablaré en el siguiente apartado

Creando algunas clases que son muy dependientes entre sí(como las listas y los productores).

Pero que a su vez permiten cierto nivel de personalización como dije antes.

4. Cosas Relativas de los apartados

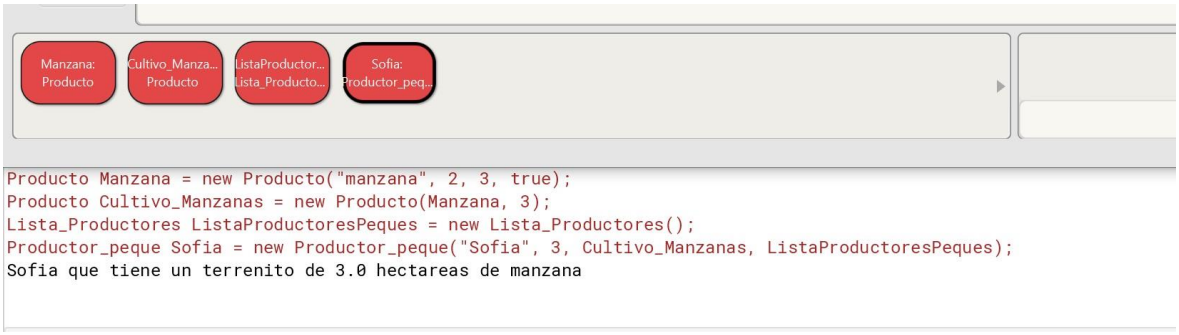
Según mi lista de prioridades tengo una cosa muy importante que no he podido implementar:

-Eso es la administración de la corporativa, es decir ,una clase que se llamase corporativa y que diera los beneficios al año ,hiciera una lista de cuantos kg le quedan por vender a cada productor, llevará el tiempo del año…Esto debido a que perdí mucho tiempo intentando implementar mi idea original de los hashmaps.

-Antes he dicho que hacer pruebas se me hacía complicado, aquí un ejemplo:

Para declarar a la productora pequeña Sofía he tenido que instanciar el producto manzana, hacerla una plantación, instanciar la lista “ListaProductoresPeques” y ponerlos como argumento.

Con esto lo que consigo es poder modificar absolutamente todo, qué lista quiero usar para el productor, Crear un cultivo sin tener granjeros… Y todas las cosas estúpidas que se te pasen por la cabeza.



Lo que me lleva al último punto:

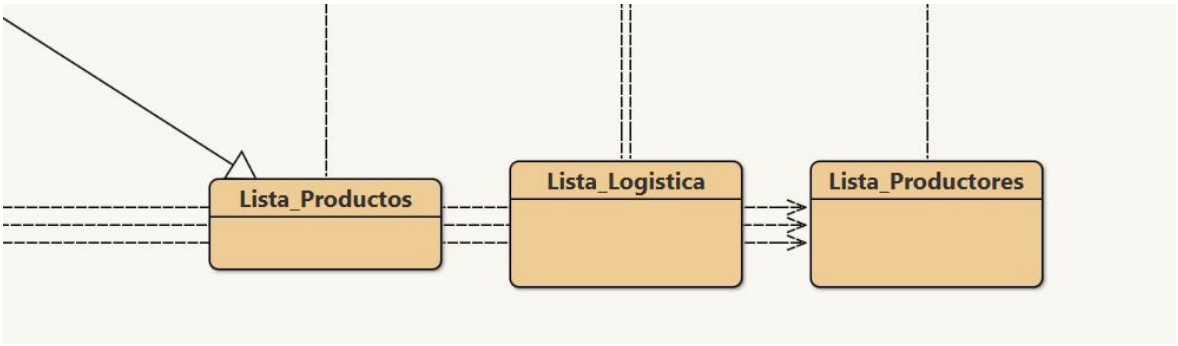
-No he conseguido un programa intuitivo y supongo que quien maneja el programa sabe usarlo ,a menudo me he visto 2 o 3 minutos declarando cosas para luego meterlas en una lista que no quería o confundir productos y liarla yo solo.

Esto es lo mismo para cada clase. Se verá más adelante en el diseño.

5. Diseño

En cuanto al diseño de mi practica decidí usar como estructura de datos el clásico Arraylist ,debido a mis fracasos con los hashmaps y a que es la estructura en la que más cómodo me encuentro.

En concreto tengo 3 listas, una por cada aspecto importante de la práctica :




```java
import java.util.*;
import java.util.ArrayList;

/**
 * lista donde se guardaran los productos
*/
public class Lista_Productos
{
    // instance variables - replace the example below with your own
    ArrayList<Producto> productos;
    /**
     * Constructor for objects of class Lista_Productos
     */
    public Lista_Productos()
    {
        productos= new ArrayList<Producto>();
    }
    /*
     * metodo para que hereden los productores
     */
    public void addProducto(Producto producto)
    {
        productos.add(producto);
    }
    //metodo que hereda los productores
      public void addProductoAGranjero(Producto producto,double hectareas)
    {
        productos.add(new Producto(producto,hectareas));
    }
      public void remove(Producto producto)
    {
        productos.remove(producto);
    }
    /**
     * Pequeño ejemplo del funcionamiento de mis productos(en este caso solo saco el primer producto)
     *
     * @param  y   a sample parameter for a method
     * @return     the sum of x and y
     */
    public void sampleMethod()
    {
        Producto primerProducto = productos.get(0);
        String nombrePrimerProducto = primerProducto.getNombre();
        double rendimientoPrimerProducto = primerProducto.getRendimiento();
        double precioPrimerProducto = primerProducto.getPrecio();
        boolean perecederoPrimerProducto = primerProducto.isPerecedero();
        // imprimir los datos del primer producto
        System.out.println("Nombre: " + nombrePrimerProducto);
        System.out.println("Rendimiento: " + rendimientoPrimerProducto);
        System.out.println("Precio: " + precioPrimerProducto);
        System.out.println("Perecedero: " + perecederoPrimerProducto);
    }
}
```

La lista de productos contiene su constructor, un método para añadir productos a la lista y otro para añadir productos a un granjero de esta forma una instancia de esta clase puede ayudarme a guardar las hectáreas cultivadas de un cierto producto por un productor o bien puede ayudarme a llevar la cuenta de cuantos productos tiene disponibles la empresa.

Por último, tiene un remove,para el caso de que el productor quiera eliminar un producto de su cosecha. Y en este caso he agregado un método de ejemplo. En el cual doy un ejemplo de como se da la dependencia entre productos y su lista ,pudiendo está acceder a cualquier elemento de productos. Esto es un síntoma de un buen diseño de clases puesto que doy una coherencia lógica a la lista, en una lista de productos ,se encuentra la información acerca de los productos.

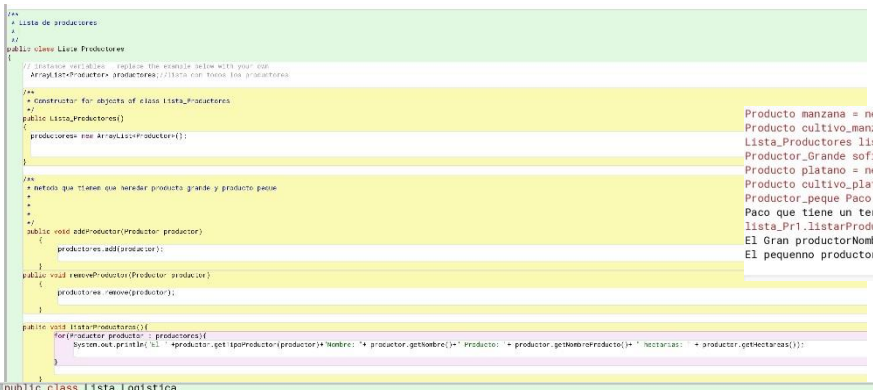
Para que se vea que todo funciona bien :

```
Producto producto1 = new Producto("manzana", 3, 3, true);
Producto producto1 = new Producto("manzana", 3, 2, true);
Lista_Productos lista_Pr1 = new Lista_Productos();
lista_Pr1.addProducto(producto1);
lista_Pr1.sampleMethod();
Nombre: manzana
Rendimiento: 3.0
Precio: 2.0
Perecedero: true
```

Luego repliqué este procedimiento a los productores y a las empresas.

Teniendo la lista de productores métodos para añadir, remover y listar productores. Y una lista de empresas que también te calcula el coste de la logística(esto me fue útil en las otras clases). *Notar que ya había agregado 2 empresas yo para tener más ejemplos ,es decir ,por default tienes mínimo 2 ofertas de empresas ,luego te puedes crear todas las empresas que tú quieras para tener más ofertas ,rebajas ,etc. Como he dicho antes todo esto de crearte las empresas es en el contexto de que al trabajador le llegue un papel con las distintas empresas y las vaya guardando en el programa. Por defecto se pueden meter 10 empresas.

El ultimo método:



Las cuales funcionan igual que la otra.

```
Producto manzana = new Producto("manzana", 2, 3, true);
Producto cultivo_manzanas = new Producto(manzana, 2);
Lista_Productores lista_Pr1 = new Lista_Productores();
Productor_Grande sofia = new Productor_Grande("Sofia", 2, cultivo_manzanas, lista_Pr1);
Producto platano = new Producto("platano", 1, 0.3, true);
Producto cultivo_platanos = new Producto(platano, 1);
Productor_peque Paco = new Productor_peque("Paco", 3, cultivo_platanos, lista_Pr1);
Paco que tiene un terrenito de 3.0 hectareas de platano
lista_Pr1.listarProductores();
El Gran productorNombre: Sofia Producto: manzana hectarias: 2.0
El pequenno productorNombre: Paco Producto: platano hectarias: 3.0
```

```
EmpresaLogistica Paco_SA = new EmpresaLogistica("PacosS.A", 43);
EmpresaLogistica Sofias_SA = new EmpresaLogistica("Sofias_SA", 23);
Lista_Logistica Lista_Empresas = new Lista_Logistica();
Lista_Empresas.addEmpresa(Paco_SA);
Lista_Empresas.addEmpresa(Sofias_SA);
Lista_Empresas.getEmpresasLogistica()
    returned Object <object reference>
```

```
if(producto.esPerecedero()) {
    while(kg >= 1000) {
        if(km >= 100) {
            coste += granLog.GranLogistica( 1000,km ,producto);//si son +100km se envia a la capital con el coste de los km

            coste += peqLog.PeqLogistica(1000,km%100);//y desde la capital se distribuye con pequeña logistica
        } else {
            coste += peqLog.PeqLogistica(km, 1000);
        }
        kg -= 1000;
    }
    if(km >= 100) {
        coste += granLog.GranLogistica(1000,km ,producto);
        coste += peqLog.PeqLogistica(kg,km%100 );
    } else {
        coste += peqLog.PeqLogistica(kg,km);
    }
} else {
    while(kg >= 1000) {
        while(km >= 50) {
            coste += granLog.GranLogistica(1000,50 ,producto);
            km -= 50;
        }
        coste += peqLog.PeqLogistica(1000,km);
        kg -= 1000;
    }
    while(km >= 50) {
        coste += granLog.GranLogistica(kg,50,producto);
        km -= 50;
    }
    coste += peqLog.PeqLogistica(kg,km);
}
return coste;
```

Utiliza los métodos que tienen cada empresa para calcular el coste de un trayecto ,en este ejemplo se siguen las directrices expresadas en la ped para realizar un cálculo con el fin de transportar 2400kg de plátanos a través de 320 km con 2 empresas(una pequeña y otra grande).

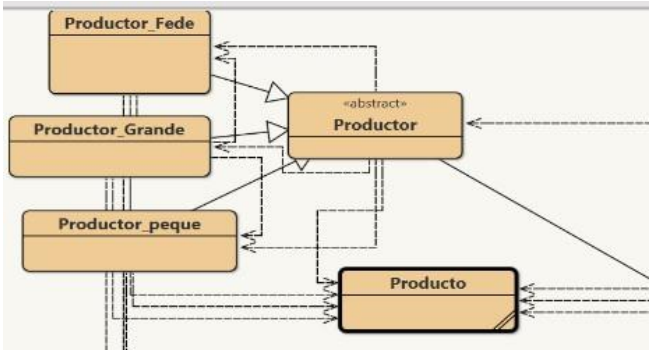
Pero de esto hablaremos después.

```
EmpresaLogistica GRADESSL = new EmpresaLogistica("GrandesSL", 32);
EmpresaLogistica peques_SA = new EmpresaLogistica("peques.SA", 12);
Producto Platano = new Producto("Platano", 4, 1, true);
Lista_Logistica.realizarLogística(Platano, 320, 2400, GRADESSL, peques_SA)
    returned double 608220.0
```

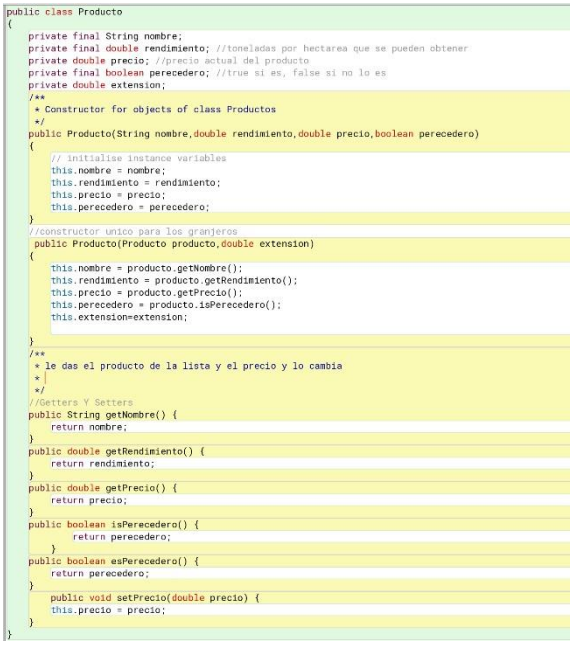
En cuanto a esos 5 lo primero que se puede ver es que productor es una clase abstracta, es la generalización de los productores, todos heredan de él y utilizan en mayor o menor medida sus constructores.

Por otro lado, decidí meter aquí a los productos ,que son el pilar de los productores, sin producto a cultivar no habría granjeros.

Ahora bien ,vayamos a una parte crucial en el sector primario ,los granjeros .Sin ellos la cooperativa no funcionaría es por ello que los explicaré a fondo.



El producto tiene unos parámetros básicos que todo producto debe tener y luego una extensión ,está es necesaria para ejecutar los constructores de los productores y o para guardar además de esos valores en la lista de productores la extensión.



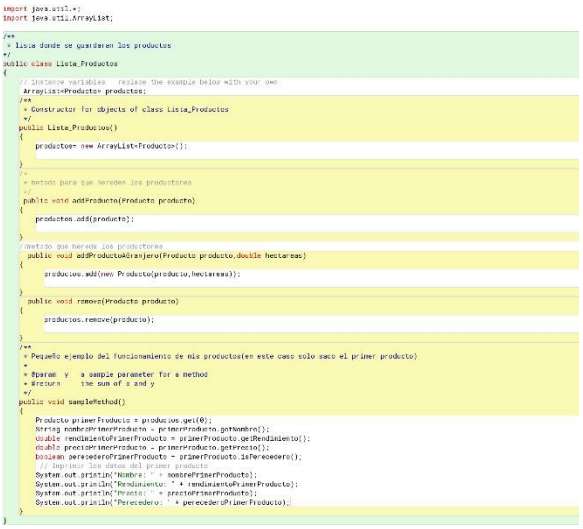
```
public class Producto
{
    private final String nombre;
    private final double rendimiento; //toneladas por hectarea que se pueden obtener
    private double precio; //precio actual del producto
    private final boolean perecedero; //true si es, false si no lo es
    private double extension;
    /**
     * Constructor for objects of class Productos
     */
    public Producto(String nombre,double rendimiento,double precio,boolean perecedero)
    {
        // initialise instance variables
        this.nombre = nombre;
        this.rendimiento = rendimiento;
        this.precio = precio;
        this.perecedero = perecedero;
    }
    //constructor unico para los granjeros
     public Producto(Producto producto,double extension)
    {
        this.nombre = producto.getNombre();
        this.rendimiento = producto.getRendimiento();
        this.precio = producto.getPrecio();
        this.perecedero = producto.isPerecedero();
        this.extension=extension;

    }
    /**
     * le das el producto de la lista y el precio y lo cambia
     *
     */
    //Getters Y Setters
    public String getNombre() {
        return nombre;
    }
    public double getRendimiento() {
        return rendimiento;
    }
    public double getPrecio() {
        return precio;
    }
    public boolean isPerecedero() {
            return perecedero;
        }
    public boolean esPerecedero() {
        return perecedero;
    }
        public void setPrecio(double precio) {
        this.precio = precio;
    }
}
```

Un ejemplo de esto lo encontramos en el segundo constructor, el cual acepta directamente un producto, esa es la forma con la que di para que en el arraylist que devuelve el productor se viera la extensión de cada producto.

Por último, tenemos una serie de getters y setters, que se usarán en otros métodos.

Destacar el método setPrecio el cual existe únicamente con la perspectiva de que tu eres un empleado que está usando el programa y que manualmente cambia el precio de cada producto según las bajadas y subidas de este mismo ,de esta manera podrás estar seguro de que el precio se actualiza al instante y no dependes de código que puede dar error en un futuro siendo más fácil hacer un mantenimiento de este.

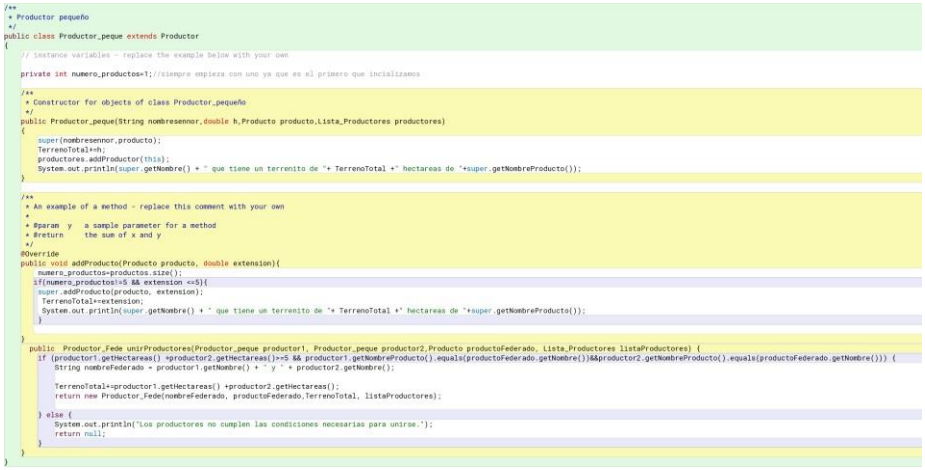
En la clase productor ,lo primero que llama la atención es que hereda de la lista de productos de forma que en los métodos addProducto y removeproducto puedo acceder a la lista de productos e ir modificándola a mi gusto, para llamar a estos métodos se harán desde los productores modificando únicamente la lista que tienen cada uno de ellos ,de forma que puede haber 2 productores con la misma lista de productos(que comparten productos) o crearte una lista para cada tipo de productor, el resto son getters que necesitaremos más adelante.



Las posibilidades son grandes, siempre y cuando no te líes tu solo con cada tipo de lista.

```
public double GetCosecha(Productor productor, Producto producto ){

    if(productor instanceof Productor_Fede){
    return productor.getHectareas()*producto.getRendimiento()*1000;//retorno el rendimiento que tiene un cosechador
}

else{
    return productor.getHectareas()*producto.getRendimiento()*1000;
}

}
```

Ahora bien, vamos a ver los tipos de productores ,aquí tenemos el productor pequeño, que hereda del padre. Tiene un constructor propio que se apoya en el del padre y 2 métodos :



addProducto añade productos a la lista que tiene el productor, solo sí no ha pasado el máximo de hectáreas y de productos.

unirProductores coge 2 productores pequeños de una lista de productores y crea un nuevo productor federado con un producto y el terreno compartido de ambos. El productor Grande es lo mismo, pero solo si tiene más de

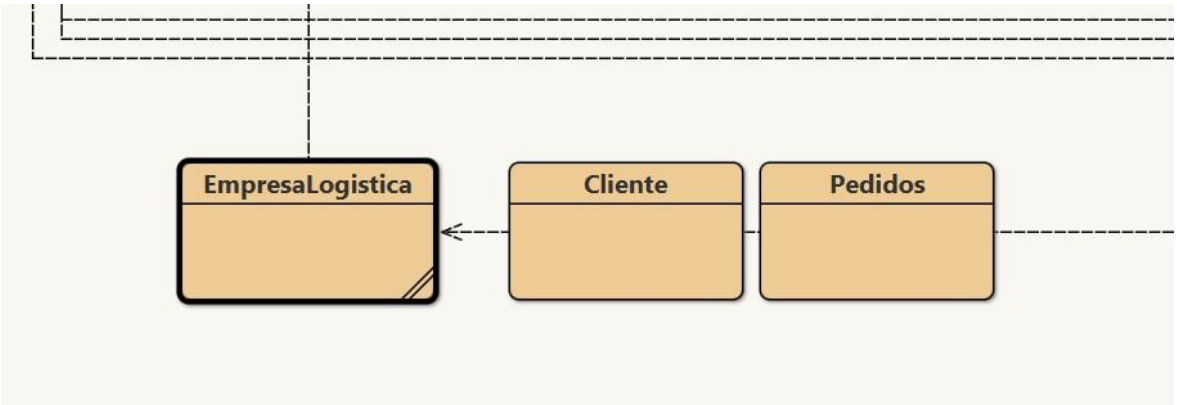
```java
public class Productor_Grande extends Productor
{
    // instance variables - replace the example below with your own



    /**
     * Constructor for objects of class Productor_Grande
     */
    public Productor_Grande(String nombresennor,double h,Producto producto,Lista_Productores productores)
    {
        super(nombresennor,producto);
          TerrenoTotal+=h;
          productores.addProductor(this);

    }

    /**
     * An example of a method - replace this comment with your own
     *
     * @param  y   a sample parameter for a method
     * @return     the sum of x and y
     */
    @Override
    public void addProducto(Producto producto, double extension){
       if(extension >=5.0){
        super.addProducto(producto, extension);
         TerrenoTotal+=extension;
         System.out.println(super.getNombre() + " que tiene un terrenito de "+ TerrenoTotal +" de "+super.getNombreProducto());
         System.out.println("done");
        }else{
        System.out.println("El productor no cumple los requisitos");
         }

    }
}
```

Y el productor federado lo mismo:

```java
public class Productor_Fede extends Productor
{
    // instance variables - replace the example below with your own
    String nombreFederado;


    /**
     * Constructor for objects of class Productor_Fede
     */
    private String producto;

    public Productor_Fede(String nombre, Producto producto,double h, Lista_Productores productores) {
          super(nombre,producto);
          TerrenoTotal+=h;
          productores.addProductor(this);

    }


}
```

Por último:

Queda la parte de la logística y venta de productos:



En Empresa Logística se hayan los métodos que necesitaremos para calcular el coste de la logística, de los que hemos hablado en el anteriormente. Además de getters necesarios para los otros métodos.

```java
public class EmpresaLogistica
{
    private String nombre;
    private double costeKm;


    /**
     * Empresa que realiza logistica
     *
     */
    public EmpresaLogistica(String n,double ckm) {
        nombre = n;
        costeKm = ckm;
    }

    /**
     * Realiza la logística de un producto y devuelve el precio de dicho servicio
     *
     */
    public double GranLogistica( int kg,int km, Producto p) {
        return (  p.getPrecio() + kg*0.5) + (km * costeKm);
    }

    /**
     * Realiza la logística de un producto y devuelve el precio de dicho servicio
     *
     */
    public double PeqLogistica( int kg,int km) {
        return  km *costeKm * kg;
    }

    //GETTER

    public String getNombre() {
        return nombre;
    }

    public double getCosteKm() {
        return costeKm;
    }
}
```

El cliente es el que te permite realizar un pedido a la corporativa, dentro de este se tiene un enumerador que dice el tipo de cliente con el que estamos tratando ,esto nos sirve para tratarle según ciertas condiciones.

Por ejemplo, si es un consumidor pagará más y podrá comprar menos. Tiene un método simple para añadir el pedido al registro y uno complejo que se encarga de comprobar que el cliente cumpla las condiciones para pedir el producto, es decir que el cliente pida menos del máximo y que haya stock disponible para dar en la empresa.

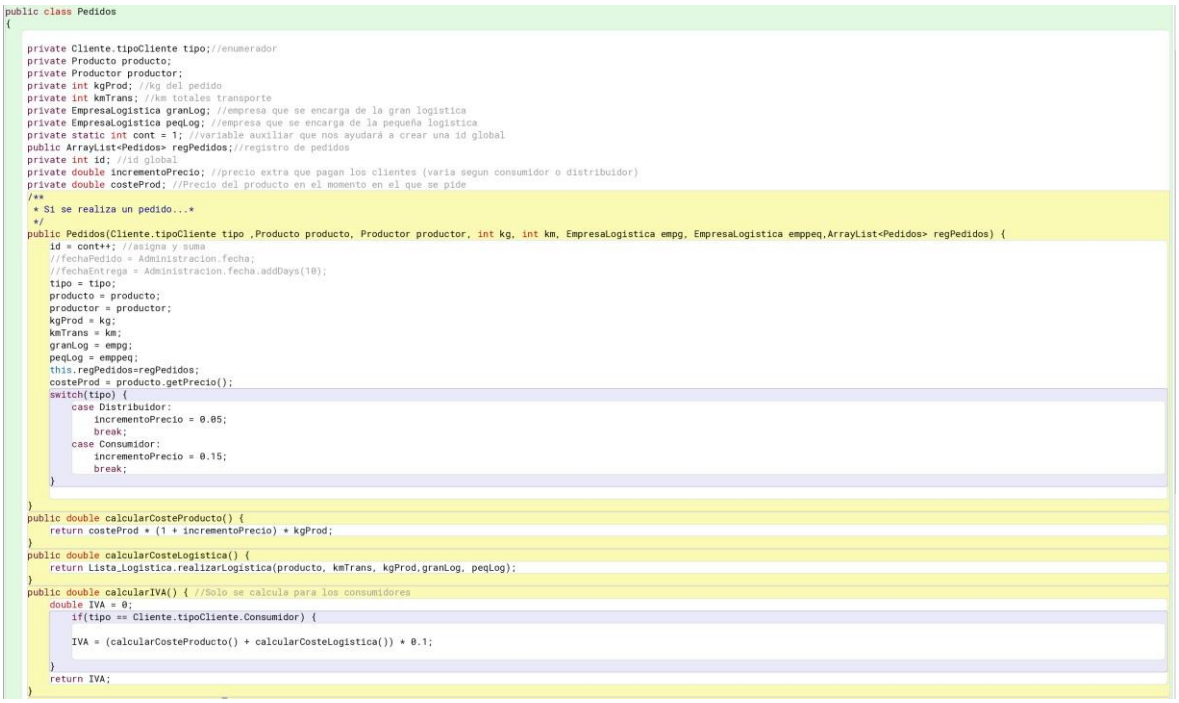
Estos datos se manejarán en la clase pedidos, y se te devolverá el pedido.

```java
public class Cliente
{
    public static enum tipoCliente
    {
        Consumidor,
        Distribuidor
    }
    tipoCliente tipo;

    String nombrecliente;

    public ArrayList<Pedidos> regPedidos;

    public Cliente(String n, tipoCliente t){
        regPedidos = new ArrayList<Pedidos>();
        nombrecliente = n;
        tipo = t;
    }
    public void addPedido(Pedidos p) {
        regPedidos.add(p);
    }
    /**
     * Realizar un pedido de parte de un cliente
     */
    public Pedidos realizarPedido(Producto producto, Productor productor, int kg, int km, EmpresaLogistica granLog, EmpresaLogistica peqLog ){
        Pedidos ped = new Pedidos( tipo,producto, productor, kg, km, granLog, peqLog,regPedidos);
        switch(tipo){

            case Consumidor:
                if(kg <= 100 && productor.GetCosecha(productor, producto)>=kg) {
                    addPedido(ped);
                } else {
                    System.out.println("Error. Un consumidor no puede pedir más de 100kg de producto ,si ha pedido menos de 100kg nos hemos quedado sin producto");
                    return null;
                }
                break;
            case Distribuidor:

                if(kg >= 1000 &&  productor.GetCosecha(productor, producto)>=kg) {
                        addPedido(ped);
                }
                else {
                        System.out.println("Un distribuidor debe pedir mínimo 1T de producto,si ha pedido más de 1T nos hemos quedado sin producto");
                        return null;
                }
                break;
            }
        return ped; //se devuelve el pedido
    }
}
```

La clase Pedidos es la más compleja con diferencia de estás 3:

En ella se puede ver el constructor que usa los datos que se les han pasado en la anterior clase y las asigna como información del pedido además de aumentar el precio un 5% si estamos hablando de un distribuidor o un 15% si es un consumidor.

A su vez se encarga de manejar todos los métodos necesarios para obtener cada elemento del paquete y mostrarlo en pantalla:

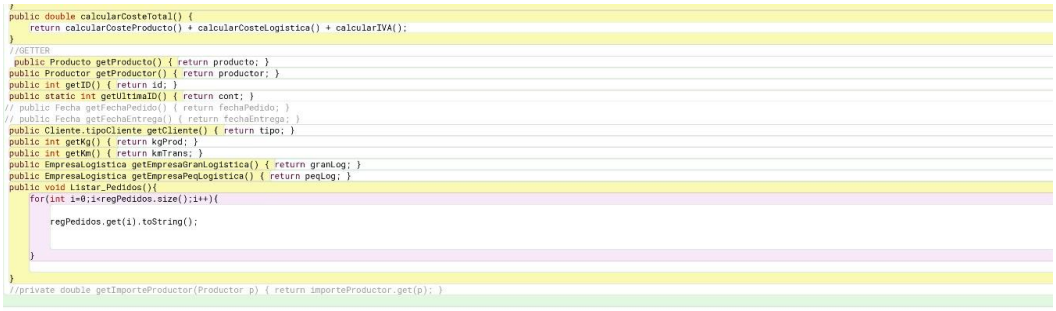


```
public class Pedidos
{

    private Cliente.tipoCliente tipo;//enumerador
    private Producto producto;
    private Productor productor;
    private int kgProd; //kg del pedido
    private int kmTrans; //km totales transporte
    private EmpresaLogistica granLog; //empresa que se encarga de la gran logistica
    private EmpresaLogistica peqLog; //empresa que se encarga de la pequeña logistica
    private static int cont = 1; //variable auxiliar que nos ayudará a crear una id global
    public ArrayList<Pedidos> regPedidos;//registro de pedidos
    private int id; //id global
    private double incrementoPrecio; //precio extra que pagan los clientes (varia segun consumidor o distribuidor)
    private double costeProd; //Precio del producto en el momento en el que se pide
    /**
     * Si se realiza un pedido...*
     */
    public Pedidos(Cliente.tipoCliente tipo ,Producto producto, Productor productor, int kg, int km, EmpresaLogistica empg, EmpresaLogistica emppeq,ArrayList<Pedidos> regPedidos) {
        id = cont++; //asigna y suma
        //fechaPedido = Administracion.fecha;
        //fechaEntrega = Administracion.fecha.addDays(10);
        tipo = tipo;
        producto = producto;
        productor = productor;
        kgProd = kg;
        kmTrans = km;
        granLog = empg;
        peqLog = emppeq;
        this.regPedidos=regPedidos;
        costeProd = producto.getPrecio();
        switch(tipo) {
            case Distribuidor:
                incrementoPrecio = 0.05;
                break;
            case Consumidor:
                incrementoPrecio = 0.15;
                break;
        }

    }
    public double calcularCosteProducto() {
        return costeProd * (1 + incrementoPrecio) * kgProd;
    }
    public double calcularCosteLogistica() {
        return Lista_Logistica.realizarLogistica(producto, kmTrans, kgProd,granLog, peqLog);
    }
    public double calcularIVA() { //Solo se calcula para los consumidores
        double IVA = 0;
            if(tipo == Cliente.tipoCliente.Consumidor) {

            IVA = (calcularCosteProducto() + calcularCosteLogistica()) * 0.1;

        }
        return IVA;
    }
```

En esta imagen se ven sus 4metodos principales:

- CalcularCosteProducto:Este método está orientado al cliente ,de esta forma podrá comparar costes.
- CalcularCosteLogistica:Igual ,la diferencia es que ya usa el método que explicamos antes en la lista de logística.
- CalcularIVA:calcula el IVA que tienen que pagar los consumidores con una simple formula. Calcula el coste total de la logística y los productos y le suma un 10% de IVA.
- CalcularCosteTotal:le suma el IVA al coste de la logística y al del producto.
- ListarPedidos:Lista todos los pedidos realizados.

El resto son un montón de getters y setters para obtener cada uno de los datos de tú pedido:



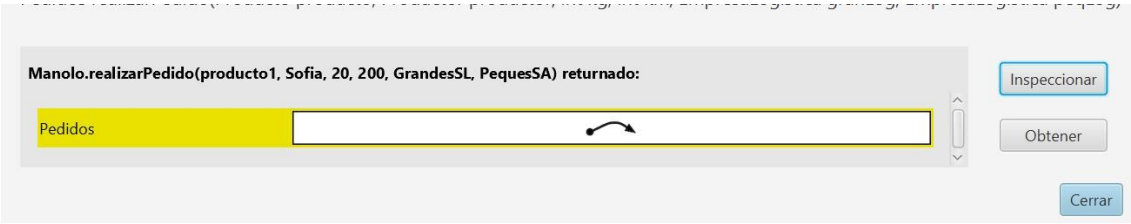
La mejor forma de ver como interactúan entre sí verlo.

```
Cliente Manolo = new Cliente("Manolo", Cliente.tipoCliente.Consumidor);
Producto producto1 = new Producto("manzana", 3, 2, true);
Lista_Productores lista_Pr1 = new Lista_Productores();
Producto Cultivomanzanas = new Producto(producto1, 20);
Productor_Grande Sofia = new Productor_Grande("Sofia", 20, Cultivomanzanas, lista_Pr1);
EmpresaLogistica PequesSA = new EmpresaLogistica("PequesSA", 20);
EmpresaLogistica GrandesSL = new EmpresaLogistica("GrandesSL", 25);
Manolo.realizarPedido(producto1, Sofia, 20, 200, GrandesSL, PequesSA)
    returned Object <object reference>
```

En la siguiente imagen se puede ver la forma en la que interactúa todo el programa entre sí.

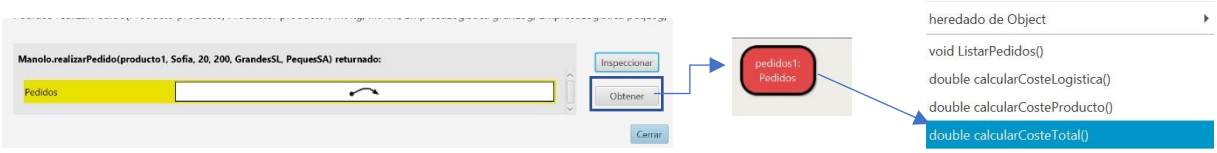
Primero declaro un cliente llamado Manolo que es de tipoConsumidor ,este cliente necesita para realizar un pedido un producto ,un productor ,los kg que va a comprar y la distancia. Declaro todo esto si es que no los tenía de antes, en este caso lo declaro de nuevo todo para que se vea bien.

Y una vez declarado Manolo puede realizar el pedido. Devolviendo un objeto pedido que tiene toda la info proporcionada.



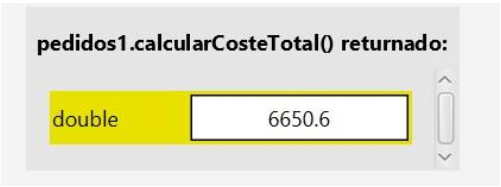
Un arraylist del producto y el productor ,otros dos para cada empresa usada y por último uno que da el registro de pedidos. Por último te devuelve las caracteristicas propias del pedido ,como su id el impuesto al precio según el tipo de cliente y el coste del producto.Como se puede ver en la propia interfaz

Es más, si le das a obtener, podrás obtener el pedido agregado:



El cual tiene todos los métodos y getters de los que he hablado antes vamos a centrarnos para este ejemplo el de calcular coste Total, que implementa el coste de logística, producto y el IVA:

Parece que en este caso la transacción costará 6650,6€.



Y esto es todo lo que tengo para dar.

6. Conclusión

Hacer este programa es fue todo un desafío, hubiera gustado implementar el Main para hacer un programa más consistente ,y hacer la clase del tiempo, pero en general estoy bastante contento con el resultado.

Es la primera de las practicas que he hecho sobre programación que sí que sabía que podría haberla implementado al 100% con un poco más de tiempo y sobre la cual me sentía en completo control, no como en C jaja.

Espero aprobarla con esto he intentado seguir todas las pautas que dijiste en la clase obligatoria y ser lo más conciso posible ,mientras explico a su vez todas las clases de manera detallada.

Por último ,pediste feedback de la clase ,me hubiera gustado que se hubiera metido más en el código de la practica en sí ,contrastar métodos y distintos tipos de almacenamiento. A parte de eso estuvo bien ,me sirvió para ideas posteriores.

Muchas gracias por ver :).