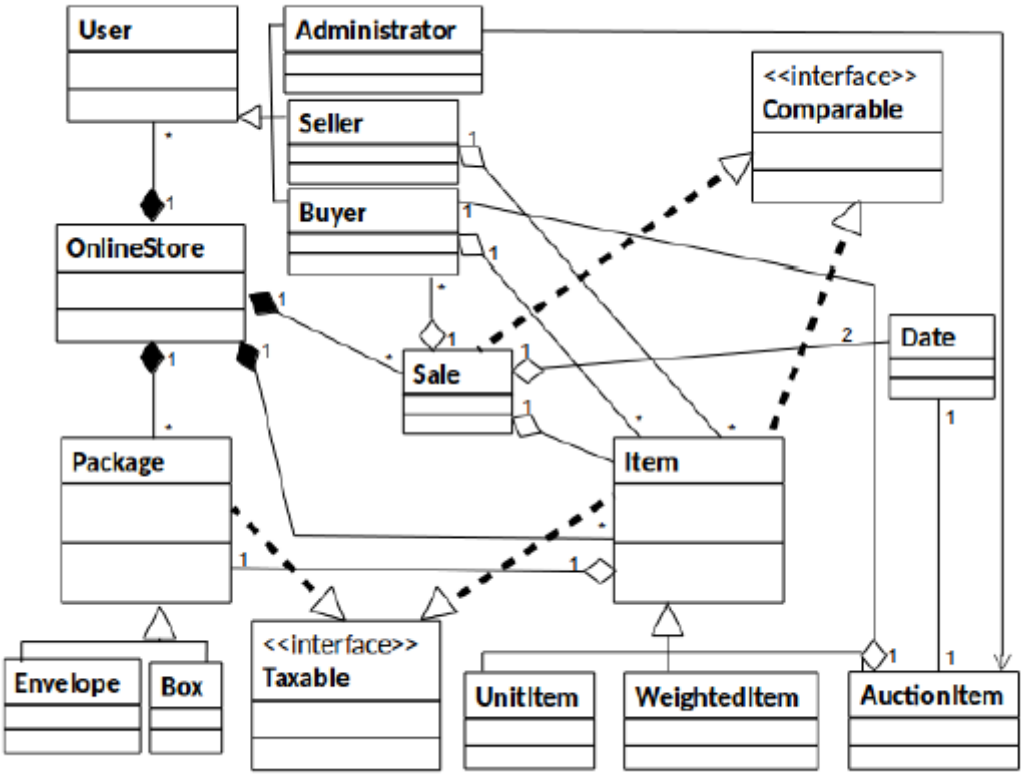
Oriol Estabanell, Albert Gubau u172749, u172750

Lab4
Programación Orientada a Objetos

-Descripción del problema a solucionar

En este laboratorio deberemos implementar el diseño realizado en el seminario 4, este diseño se basa en una tienda online (**Online Store**) en la que encontramos diferentes clases que mantienen relaciones de herencia con otras y nos permiten implementar correctamente como funcionaría una tienda online básica.

Este diseño se basará en una ampliación del diseño del laboratorio 3 en el que tendremos:



Como podemos ver, hemos añadido dos nuevas clases (**Date y Sale**) y dos interfaces(**Taxable y Comparable**).

Por lo tanto, deberemos modificar nuestra **tienda online** para implementar de manera correcta estas clases e interfaces. Se nos pide que:

Modifiquemos el **main** para tener un método que procese las ventas (**sell**) y un método que actualice el día en el que estamos (**dayPass**). La clase **OnlineStore** deberá tener almacenadas las **ventas** y el **día en el que estamos**.

Implementemos la interfaz **Taxable** en **Item** y **Package**.

Ordenemos los **items** por precio y las ventas por fecha (implementación de **Comparable**).

Oriol Estabanell, Albert Gubau u172749, u172750

- Modificaciones en el main (OnlineStore)

- Añadimos atributos nuevos

En nuestro **main** deberemos añadir un atributo que tenga en cuenta el día en el que nos encontramos y otro atributo que mantenga un **registro de todas las ventas realizadas**.

Para poder añadir de manera correcta estos atributos hemos pensado en tener un atributo de tipo Date que sea la fecha actual al que llamamos **currentDate** y una **LinkedList de ventas**.

Para poder implementar estos atributos, primero deberemos definir las clases **Date** y **Sale**:

- Clase Date

Como sabemos por nuestro diagrama, la clase **Date** implementará la interfaz **Comparable**, por lo que tendrá que redefinir el método **compareTo** para poder ordenar las ventas por fecha, pero eso lo veremos más adelante. Nuestra clase **Date** la implementaremos de la siguiente manera:

```
public class Date implements Comparable{

    private int day;
    private int month;
    private int year;

    Date(int d, int m, int y){
        day = d;
        month = m;
        year = y;
    }

    public int getDay(){...3 lines }
    public int getMonth(){...3 lines }
    public int getYear(){...3 lines }
    public void setDay(int d){...3 lines }
    public void setMonth(int m){...3 lines }
    public void setYear(int y){...3 lines }

    @Override
    public int compareTo(Object o){

        if(o instanceof Date){

            Date date = (Date)o;

            if(year < date.getYear()){
                return -1;
            }
            else if(year > date.getYear()){
                return 1;
            }
            else if(month < date.getMonth()){
                return -1;
            }
            else if (month > date.getMonth()){
                return 1;
            }
            else if(day < date.getDay()){
                return -1;
            }
            else if (day > date.getDay()){
                return 1;
            }
        }
        return 0;
    }
}
```

Nuestra clase Date tendrá tres atributos (**day, month, year**), podemos observar su método constructor y sus métodos **getters** y **setters**.

En esta imagen tenemos ocultos los getters y los **setters**, estos métodos no tienen mayor complicación que la que ya hemos visto en anteriores prácticas, los **getters** devuelven el atributo que indica su nombre y los setters modifican el valor del atributo por el que entra por parámetro del método.

Debajo de los getters y setters de la clase podemos ver la redefinición del método **compareTo** de la interfaz Comparable en la que se devuelve -1 si la fecha con la que comparamos es mayor que en la que estamos, 0 si es la misma fecha y 1 si la fecha en la que estamos es mayor que la fecha con la que la comparamos.

La implementación de este método nos permitirá en un futuro ordenar las ventas por su fecha de realización.

- Clase Sale

Como sabemos por nuestro diagrama, la clase **Sale** implementará la interfaz **Comparable**, por lo que tendrá que redefinir el método **compareTo** para poder ordenar las ventas por fecha. Nuestra clase Sale la implementaremos de la siguiente manera:

```
public class Sale implements Comparable{

    private Item item;
    private Buyer buyer;
    private Date saleDate;
    private Date shippingDate;
    private Package pack;

    public Sale(Item i, Buyer b, Date sd, Date shd, Package p){

        item = i;
        buyer = b;
        saleDate = sd;
        shippingDate = shd;
        pack = p;
    }

    public Item getItem(){...3 lines }

    public Buyer getBuyer(){...3 lines }

    public Date getDate(){...3 lines }

    public Date getShippingDate(){...3 lines }

    public void setSaleDate(Date d){...3 lines }

    public void setShippingDate(Date d){...3 lines }

    @Override
    public int compareTo(Object o){

        if(o instanceof Sale){

            Sale s = (Sale)o;
            return s.getDate().compareTo(getDate());
        }
        return 5;
    }
}
```

En nuestra clase Sale declaramos 5 atributos referentes al **item** que se vende, el comprador que lo compra, la fecha de compra, la fecha de envío y el empaquetado del item que se vende.

Primeramente, podemos ver su método **constructor** y sus respectivos **getters y setters** que los mostramos ocultos porque sabemos como funcionan y no presentan ninguna novedad con respecto a los getters y setters que hemos visto en anteriores prácticas.

Por último, encontramos la implementación del método **compareTo** de la interfaz **Comparable** en el que llamamos al compareTo de la fecha para poder ordenar las ventas por su fecha.

Una vez hemos creado las dos clases necesarias para las modificaciones de nuestro main de la clase OnlineStore, ya podemos empezar a añadir los atributos nuevos de manera que tendremos:

```
public class OnlineStore {

    public static LinkedList< Item > itemsAvailable;
    public static LinkedList< Item > itemsSold;
    public static LinkedList< User > users;
    public static LinkedList< Package > packages;
    public static LinkedList<Sale> sales;
    public static Date currentDate;
    public static double totalPrice;
    public static double totalTaxes;
    public static double totalProfit;

    public static void init(){

        itemsAvailable = new LinkedList< Item >();
        itemsSold = new LinkedList< Item >();
        users = new LinkedList< User >();
        packages = new LinkedList< Package >();
        sales = new LinkedList< Sale >();
        currentDate = new Date(29,10,2020);
        totalPrice = 0.0;
        totalTaxes = 0.0;
        totalProfit = 0.0;
    }
```

Podemos ver como hemos añadido un atributo de tipo fecha que tiene en cuenta el día en el que nos encontramos y una lista de ventas para mantener un registro de estas.

También hemos añadido un atributo **TotalTaxes** que nos servirá para saber el total de impuestos que hemos pagado, pero eso lo veremos más adelante.

En el método **init** declaramos que el inicio de nuestra **OnlineStore** es el da **29/11/2020**.

- Añadimos el método sell y el método dayPass

También se nos pide añadir el método sell y el método **dayPass**, estos métodos nos servirán para procesar correctamente las ventas de nuestra tienda online y para actualizar el día en el que nos encontramos respectivamente.

- Método sell

Este método deberá vender los artículos de nuestra tienda online de manera que cuando venda un artículo cree una nueva instancia de **Sale** y la añada a la lista de ventas que tenemos como atributo de la tienda online. También calculara el **precio**, los **beneficios** y los **impuestos**, y aplicará el método **buy** y el método **sell** del **Buyer** y el **Seller** respectivamente:

```
public static void sell(Item item, Buyer b, Seller s){

    b.buy(item);

    totalPrice += item.getPrice();
    totalTaxes += item.getPriceOnlyTax();


    if(item instanceof UnitItem){
        ((UnitItem)item).sell(0);

    }else if(item instanceof WeightedItem){
        ((WeightedItem)item).sell(0.0);
    }

    s.sell(item);

    int saleday = currentDate.getDay();
    int salemonth = currentDate.getMonth();
    int saleyear = currentDate.getYear();

    Date saledate = new Date(saleday, salemonth, saleyear);
    Date shippingdate = saledate;
    Sale sale = new Sale(item, b, saledate, shippingdate, item.getPackage());

    sales.add(sale);

    totalProfit += item.calculateProfit();

    itemsSold.add(item);

    itemsAvailable.remove(item);
}
```

Como podemos ver, el método **sell** toma por parámetro el **Buyer**, el **Seller** y el item que vendemos. Primero aplicamos el método buy del **Buyer**, después aumentamos el precio total y los impuestos totales, y aplicamos el sell según el tipo de item que estamos vendiendo, seguidamente aplicamos el método sell del **Seller**, después creamos una nueva instancia de Sale con los parámetros necesarios según nuestra implementación de la clase Sale y por último, añadimos la venta a la lista de ventas, aumentamos el beneficio total, añadimos el item vendido a la lista de items vendidos y eliminamos el item de la lista de items disponibles.

- Método dayPass

Este método lo que hará será actualizar el día en el que estamos y si el nuevo dia coincide con la fecha de finalización de una subasta procesará la venta del artículo que se estaba subastando. Este método lo implementamos de la siguiente manera:

```
public static void dayPass(){

    int currentDay = currentDate.getDay();
    int currentMonth = currentDate.getMonth();
    int currentYear = currentDate.getYear();

    currentDay++;

    if(currentDay==32){

        currentDay = 1;
        currentMonth++;

        if(currentMonth == 13){

            currentMonth = 1;
            currentYear++;
        }
    }

    currentDate.setDay(currentDay);
    currentDate.setMonth(currentMonth);
    currentDate.setYear(currentYear);

    for(Item i: itemsAvailable){

        if(i instanceof AuctionItem){

            if(((AuctionItem)i).getDeadline().compareTo(currentDate) == 0){

                for(User u:users){

                    if(u instanceof Seller){
                        sell(i, ((AuctionItem)i).getBuyer(), (Seller)u);
                    }
                }
            }
        }
    }
}
```

Como podemos observar, tomamos la fecha actual y la actualizamos teniendo en cuenta los años los meses y los días. En nuestra implementación, todos los meses acaban el día **31**, de manera que si sobrepasamos ese día se pasa al día **1** del siguiente mes y con los años algo similar pero acabando en el mes **12**.

Después de actualizar el día recorremos la lista de **items** disponibles en busca de los items de subasta, cuando encontramos uno comparamos la nueva fecha con la de la finalización de la puja del **item** y si coinciden procesamos la venta con el método **sell** que hemos visto anteriormente.

- Implementación de la interfaz Taxable en Item y Package

En esta práctica se nos presenta el concepto de que tanto los elementos que vendemos como los empaquetados tienen un coste y unos impuestos para la OnlineStore, de manera que deberemos tenerlos en cuenta implementando una interfaz llamada **Taxable**:

```
public interface Taxable {

    public static final double iva = 0.21;

    public double getPrice();

    public double calculateProfit();

    public double getPriceOnlyTax();

    public double getPricePlusTax();
}
```

Como podemos ver, en la interfaz **Taxable**, tenemos un valor **estático** correspondiente al **IVA** que sera del **21% (0.21)**, un método para tomar el precio, el **beneficio**, el precio del **impuesto** y el **precio añadiendo los impuestos**. Esta interfaz podría tener el método **sumTotalTax** pero hemos decidido prescindir de este dado que un profesor nos permitió hacerlo.

- Implementación de Taxable en Package

Como hemos dicho, los empaquetados en nuestra modificación tendrán un precio, así que, deberemos añadir dos atributos nuevos, el coste y el precio del empaquetado, para poder implementar la interfaz. Como la implementación de la interfaz dependerá del tipo de empaquetado en el que nos encontramos deberemos implementarla en las subclases de **Package**, de manera que en la misma clase Package declararemos los métodos de **Taxable** como abstractos y la clase Package pasará a ser abstracta, de este modo redefiniremos los métodos de **Taxable** en las subclases de **Package** y no en la misma clase Package:

```
public abstract class Package implements Taxable{

    private double width;
    private double height;

    public Package(double w, double h){
        width = w;
        height = h;
    }


    public double getWidth(){...3 lines }

    public double getHeight(){...3 lines }

    public void setWidth(double w){...3 lines }

    public void setHeight(double h){...3 lines }

    @Override
    public abstract double getPrice();


    @Override
    public abstract double calculateProfit();


    @Override
    public abstract double getPriceOnlyTax();


    @Override
    public abstract double getPricePlusTax();
}
```

Una vez hemos mostrado nuestra modificación de la clase **Package**, así serán las modificaciones de **Envelope**:

```java
public class Envelope extends Package {

    private String name;
    private double price;
    private double cost;

    public Envelope(int w, int h, String n, double c, double p){
        super(w, h);
        name = n;
        price = p;
        cost = c;
    }

    public String getName(){...3 lines }

    @Override
    public double getPrice(){
        return price;
    }

    public void setPrice(double p){
        price = p;
    }

    public double getCost(){
        return cost;
    }

    public void setCost(double c){
        cost = c;
    }

    @Override
    public double calculateProfit(){
        return getPrice()-getCost();
    }

    public void setName(String n){...3 lines }
    public Boolean isSuitable(double[] size){...4 lines }

    @Override
    public double getPriceOnlyTax(){
        return getPrice()*iva;
    }


    @Override
    public double getPricePlusTax(){
        return getPrice() + getPriceOnlyTax();
    }
}
```

Como podemos observar, añadimos dos atributos nuevos, el precio y el coste del envoltorio, estos atributos vendrán dados por el parámetro del constructor así que ya dependerán del tamaño del empaquetado.

Dado que hemos añadido dos nuevos atributos, añadimos también sus **getters y setters**. Por último, redefinimos los métodos de la interfaz **Taxable** para poder calcular el precio sin impuesto, el beneficio, el precio de los impuestos y el precio final con los impuestos añadidos.

Vemos que los métodos que ya han sido explicados están en oculto debido a que no hemos modificado ningún aspecto de estos.

Una vez hemos mostrado nuestra modificación de la clase **Package**, así serán las modificaciones de **Box**:

```java
public class Box extends Package{

    private double depth;
    private double price;
    private double cost;

    public Box(int w, int h, int d, double c, double p){
        super(w,h);
        depth = d;
        cost = c;
        price = p;
    }


    public double getDepth(){...3 lines }

    @Override
    public double getPrice(){
        return price;
    }

    public void setPrice(double p){
        price = p;
    }

    public double getCost(){
        return cost;
    }

    public void setCost(double c){
        cost = c;
    }

    @Override
    public double calculateProfit(){
        return getPrice()-getCost();
    }

    public void setDepth(double d){...3 lines }
    public Boolean isSuitable(double[] size){...28 lines }

    @Override
    public double getPriceOnlyTax(){
        return getPrice()*iva;
    }


    @Override
    public double getPricePlusTax(){
        return getPrice() + getPriceOnlyTax();
    }
}
```

Como podemos observar, hemos añadido los nuevos atributos, coste y precio, como en el caso de **Envelope**, estos atributos dependerán del tamaño del envoltorio que entremos por parámetro del método constructor de la clase.

Vemos que tenemos los **getters** y los **setters** correspondientes a los nuevos atributos y la implementación de los métodos de **Taxable**, de manera que se puede devolver el precio sin impuestos, los beneficios, el precio de los impuestos y el precio con los impuestos.

Los métodos ocultos no añaden ninguna modificación con respecto a la práctica anterior, por este motivo no se muestran en este informe.

- Implementación de Taxable en Item

De la misma manera que la clase Package, la clase **Item** implementará la interfaz **Taxable**. A diferencia de la clase Package, la clase Item también implementará la interfaz Comparable, que le permitirá comparar objetos de tipo Item para posteriormente ordenarlos según su precio. Las modificaciones en la clase Item serán las mismas que en la clase Package, pasaremos a declarar la clase como abstracta y los métodos de **Taxable** los declararemos como abstractos para poder implementar-los en las subclases de **Item**:

En la clase **Item** no nos hará falta añadir el precio y el coste debido a que ya se encontraban en ésta por la implementación de la práctica anterior.

Como podemos ver, tenemos los métodos ocultos de la anterior práctica y los métodos declarados como abstractos de la interfaz **Taxable**. Como hemos mencionado antes, al tener los métodos de Taxable declarados como abstractos y no implementarlos, la clase Item pasa a ser abstracta.

También podemos ver un método que aún no hemos explicado, el método **compareTo** de la clase Item, más adelante lo explicaremos, este método nos permite comparar items según su precio.

Una vez hemos mostrado nuestra modificación de la clase **Item**, así serán las modificaciones de **UnitItem**:

```java
public class UnitItem extends Item{

    private double unitPrice;
    private int quantity;
    private int quantityRemaining;


    UnitItem(String n, String t, double[] s, double c, double uprice, int q){...8 lines }

    public double getQuantityRemaining(){...4 lines }

    @Override
    public double getPrice(){

        return unitPrice*quantityRemaining;
    }

    @Override
    public double calculateProfit(){

        return (quantity-quantityRemaining)*(unitPrice-getCost())- getPackage().getPricePlusTax();
    }

    public double sell(int q){...6 lines }

    @Override
    public double getPriceOnlyTax(){
        return getPrice()*iva;
    }


    @Override
    public double getPricePlusTax(){
        return getPrice() + getPriceOnlyTax();
    }
}
```

Como podemos ver, implementamos los métodos de la interfaz **Taxable**, de manera que, el método **getPrice** se queda igual que en la práctica anterior y al método **calculateProfit** añadiremos el hecho de que los empaquetados tienen un precio para la OnlineStore, de manera que este coste por el paquete se le restará al beneficio por el **item**.
Seguidamente, podemos ver la implementación de **getPriceOnlyTax** y **getPricePlusTax** que devolverán el precio del impuesto y el precio total del elemento más los impuestos.

En cuanto a los dos tipos de **items** siguientes, veremos que tienen una implementación bastante parecida con pocas modificaciones para que puedan implementar la interfaz **Taxable** de manera correcta.

Una vez hemos mostrado nuestra modificación de la clase **Item**, así serán las modificaciones de **WeightedItem**:

```
public class WeightedItem extends Item{

    private double pricePerWeight;
    private double weight;
    private double weightRemaining;

    public WeightedItem(String n, String t, double[] s, double c, double wprice, double w){...8 lines }

    public double getWeightRemaining(){...4 lines }

    @Override
    public double getPrice(){

        return pricePerWeight*weightRemaining;
    }


    @Override
    public double calculateProfit(){

        return (weight-weightRemaining)*(pricePerWeight-getCost()) -getPackage().getPricePlusTax();
    }


    public double sell(double w){...6 lines }

    @Override
    public double getPriceOnlyTax(){
        return getPrice()*iva;
    }


    @Override
    public double getPricePlusTax(){
        return getPrice() + getPriceOnlyTax();
    }
}
```

Como podemos ver, lo que haremos será lo mismo que en la subclase UnitItem, adaptaremos los métodos de Taxable a los atributos de la clase, en este caso, la clase de Items que se compran por peso.

Como vemos, implementamos los métodos de la interfaz **Taxable**, de manera que, el método **getPrice** se queda igual que en la práctica anterior y al método **calculateProfit** añadiremos el hecho de que los empaquetados tienen un precio para la OnlineStore, de manera que este coste por el paquete se le restará al beneficio por el **item**.
Seguidamente, podemos ver la implementación de **getPriceOnlyTax** y **getPricePlusTax** que devolverán el precio del impuesto y el precio total del elemento más los impuestos.

Una vez hemos mostrado nuestra modificación de la clase **Item**, así serán las modificaciones de **AuctionItem**:

```java
public class AuctionItem extends Item{ //AcutionItem es una clase hija de Item, por lo tanto, usamos e

    private double currentPrice;
    private Buyer bidder;
    private Date deadline;
    public static final int fee = 5;
    public static final double percent = 0.05;

    AuctionItem(String n, String t, double[] s, double c, double startingPrice, Date d){...7 lines }

    @Override
    public double getPrice(){
        return currentPrice;
    }

    @Override
    public double calculateProfit(){

        if(bidder == null ){

            return fee;
        }
        return fee + getPrice()*percent - getPackage().getPricePlusTax();
    }

    public void makeBid(Buyer b, double p){

        if(p > currentPrice){

            bidder = b;
            currentPrice = p;

            System.out.println(bidder.getName() + " has bid " + currentPrice + " for the " + getName()
        }
    }

    public Boolean frozen(Date d){

        if(d.compareTo(deadline)<0){
            return false;
        }
        return true;
    }
```

Para empezar, podemos ver que el atributo **deadline** es de tipo Date gracias a la implementación de nuestra clase Date. Vemos que por el hecho anterior, nuestro método **frozen** se hace más sencillo puesto que aplicamos el método **compareTo** que hemos visto anteriormente.

Por lo demás, vemos que los métodos de la práctica anterior se mantienen y añadimos solo la resta del precio del empaquetado para calcular el beneficio.

```java
@Override
public double getPriceOnlyTax(){
    return getPrice()*iva;
}


@Override
public double getPricePlusTax(){
    return getPrice() + getPriceOnlyTax();
}
```

Aquí podemos ver como implementamos los métodos **getPriceOnlyTax** y **getPricePlusTax** para la clase **AuctionItem**. Estas implementaciones son prácticamente idénticas para las 3 subclases de **Item**.

- Ordenar los items por precio y las ventas por fecha

Como hemos visto en anteriores apartados, la clase **Date, Item y Sale**, implementan la interfaz Comparable, de manera que se pueden comparar dos instancias de estas clase para poder ordenarlos.

Hemos visto la implementación de Comparable en Date y Sale, pero no la de la clase Item. La implementación de la interfaz Comparable en la clase Item es la siguiente:

```
@Override
public int compareTo(Object o){

    if(o instanceof UnitItem){

        UnitItem i = (UnitItem)o;

        if(i.getPrice() - getPrice() > 0){
            return -1;
        }
        if(i.getPrice() - getPrice() == 0){
            return 0;
        }
        if(i.getPrice() - getPrice() < 0){
            return 1;
        }
    }

    if(o instanceof WeightedItem){

        WeightedItem i = (WeightedItem)o;

        if(i.getPrice() - getPrice() > 0){
            return -1;
        }
        if(i.getPrice() - getPrice() == 0){
            return 0;
        }
        if(i.getPrice() - getPrice() < 0){
            return 1;
        }
    }

    if(o instanceof AuctionItem){

        AuctionItem i = (AuctionItem)o;

        if(i.getPrice() - getPrice() > 0){
            return -1;
        }
        if(i.getPrice() - getPrice() == 0){
            return 0;
        }
        if(i.getPrice() - getPrice() < 0){
            return 1;
        }
    }

    return 5;
}
```

Como podemos observar, redefinimos el método **compareTo** de la interfaz **Comparable**. Este método tal y como lo hemos redefinido, nos devuelve -1 si el precio con el que estamos comparando es mayor, 0 si los precios son iguales y 1 si el precio con el que estamos comparando es menor que el de nuestro item.

Como vemos, tenemos una implementación para cada tipo de item, ya sea **UnitItem, WeightedItem o AuctionItem**, esto es debido a que cada tipo de item tiene una manera de calcular el precio distinta y así nos aseguramos de que cada uno compara correctamente los precios.

También podríamos haber definido el método **compareTo** como abstracto en la clase Item y después redefinirlo en las subclases de Item pero en nuestro caso lo hemos hecho de esta manera.

Algo a tener en cuenta es que si encuentra un elemento cuyos precios dan valores diferentes al ser comparados, se devuelve un 5, es un número escogido arbitrariamente por nosotros y podríamos decir que sería nuestro código de ERROR.

Una vez ya hemos visto todas las implementaciones de la interfaz **Comparable** en las diferentes clases que la implementan, podemos ver como ordenaremos los items por precio y las ventas por fecha.

En cuanto a los items, lo que haremos será aplicar el método **sort** de la clase **Collections**, de manera que este usará los métodos de **compareTo** que hemos implementado para ordenar los items de la tienda por precio dado que hemos definido el compareTo de los items de manera que los compare por precio.

```
System.out.println("\nSORTED LIST OF ITEMS IN INCREASING PRICE ORDER WITHOUT IVA:\n");

Collections.sort(itemsAvailable);

for(Item i: itemsAvailable){
    System.out.println("We have the item " + i.getName() + " with price: " + i.getPrice());
}

System.out.println("\nSORTED LIST OF ITEMS IN INCREASING PRICE ORDER WITH IVA:\n");

for(Item i: itemsAvailable){
    System.out.println("We have the item " + i.getName() + " with price: " + i.getPricePlusTax());
}
```

Collections.sort ordenará los **items** de la tienda online en orden ascendente e imprimiremos por pantalla los items ordenados con sus precios sin IVA y los **items** ordenados con los precios con **IVA** ayudándonos del método **getPricePlusTax()**.

En cuanto a las ventas, aplicaremos algo parecido de manera que aplicaremos el método **sort** en la lista de ventas (sales) y luego imprimiremos por pantalla el resultado de haberlas ordenado. Dado que en la clase Sale hemos implementado **compareTo** de manera que compare las ventas por fecha, el método **sort** ordenará las ventas por fecha en orden de más actual a más vieja.

```
System.out.println("\nSALES SORTED BY DATE STARTING BY THE LAST ITEM SOLD:\n");

Collections.sort(sales);

for(Sale s: sales){
    Date date = s.getDate();
    System.out.println("We sold the item " + s.getItem().getName() + " the day " +
}
System.out.println("\n");
```

Al realizar estas dos implementaciones obtendremos la salida por pantalla:

```
SORTED LIST OF ITEMS IN INCREASING PRICE ORDER WITHOUT IVA:

We have the item Rice with price: 125.0
We have the item Gaming Chair with price: 1000.0
We have the item TV with price: 4000.0

SORTED LIST OF ITEMS IN INCREASING PRICE ORDER WITH IVA:

We have the item Rice with price: 151.25
We have the item Gaming Chair with price: 1210.0
We have the item TV with price: 4840.0
```

```
SALES SORTED BY DATE STARTING BY THE LAST ITEM SOLD:

We sold the item Armario the day 11/11/2020.
We sold the item TV the day 3/11/2020.
We sold the item Gaming Chair the day 2/11/2020.
We sold the item Rice the day 1/11/2020.
```

- Algunos aspectos no mencionados

En cuanto al contador de impuestos totales, este lo actualizaremos cada vez que se realice una venta, de manera que en la ejecución de nuestro programa obtendremos el número de impuestos pagados totales:

```
Total price without IVA: 18125.0
Total taxes: 3806.25
Total profit: 2704.4797
```

El método **dayPass()**, lo aplicaremos en el main para ir avanzando de día, de manera que la **tienda online** se vaya actualizando al día en el que estamos. Algunos ejemplos de la aplicación de **dayPass**:

```
if(!auctionitem.frozen(currentDate)){
    auctionitem.makeBid((Buyer)users.get(1), 10500.0);
}

dayPass(); //7/11/2020

if(!auctionitem.frozen(currentDate)){
    auctionitem.makeBid((Buyer)users.get(0), 11000.0);
}

dayPass(); //8/11/2020
dayPass(); //9/11/2020
dayPass(); //10/11/2020

if(!auctionitem.frozen(currentDate)){
    auctionitem.makeBid((Buyer)users.get(2), 12000.0);
}
```

```
for(int i = 0; i<itemsAvailable.size(); i++){

    Item item = itemsAvailable.get(i); //TOMAM
    Buyer buyer = (Buyer)users.get(i);

    sell(item, buyer, seller); //APLICAMOS EL
    dayPass(); //2/11/2020
    i--; //AHORA EL ITEM BORRADO HA PUESTO EL
}

dayPass(); //5/11/2020
dayPass(); //6/11/2020
```

- Ejecución del programa

Algunos aspectos de la ejecución del programa que podemos notar serían:

```
PACKAGE ASSIGNMENT FOR THE ITEMS AVAILABLE:

Box with size (100.0, 150.0, 300.0) and price 0.1452 assigned to item Gaming Chair.
Envelope A4 with price 0.0484 assigned to item Rice.
Box with size (10.0, 100.0, 100.0) and price 0.12100000000000001 assigned to item TV.
```

La asignación del paquete con su precio.

```
USERS SHOPPING:

Marcos Navajas is buying item Rice for 125.0 (price without IVA) and the price with IVA is  151.25 euros.
Price 151.25 is getting charged into account 12345678 from user Marcos Navajas.
Daniel Leorri has sold item Rice and 49.9516 euros are the benefits obtained for the item.
Marcos Navajas is buying item Gaming Chair for 1000.0 (price without IVA) and the price with IVA is  1210.0 euros.
Price 1210.0 is getting charged into account 12345678 from user Marcos Navajas.
Daniel Leorri has sold item Gaming Chair and 399.8548 euros are the benefits obtained for the item.
Marcos Navajas is buying item TV for 4000.0 (price without IVA) and the price with IVA is  4840.0 euros.
Price 4840.0 is getting charged into account 12345678 from user Marcos Navajas.
Daniel Leorri has sold item TV and 1599.879 euros are the benefits obtained for the item.
```

Los usuarios comprando.

Oriol Estabanell, Albert Gubau u172749, u172750

- Conclusiones, soluciones alternativas y contribuciones

Las conclusiones que sacamos de esta práctica son muy parecidas a las conclusiones que sacamos en la práctica anterior, es una práctica que tiene muchos vacíos en cuanto al funcionamiento de algunos métodos y nos ha parecido que no se ha implementado de la mejor manera por parte en los enunciados. Si a estos aspectos le añades el poco tiempo del que hemos dispuesto y el poco **feedback** recibido, obtenemos resultados que son buenos pero podrían ser mucho mejores.

Algunas soluciones alternativas que proponemos serían las de no implementar la interfaz **Taxable** y asumir que los precios de la tienda ya tienen en cuenta el IVA en cada uno de sus productos y el dinero gastado en su empaquetado. De manera que la tienda online simplemente funcionaría como en la práctica anterior y no deberíamos realizar la implementación de la interfaz, sino que deberiamos descomponer el precio de cada item en impuestos y beneficios.

En esta práctica, **Oriol Estabanell** ha realizado la parte de la implementación de la interfaz Taxable en las clases Item y **Package**, y los aspectos relacionados con esta implementación en el main de la clase **OnlineStore**. **Albert Gubau** ha realizado la parte de ordenar los items por precio y las ventas por fecha así como la implementación de la interfaz Comparable en las clases Date, Sale e **Item**.
De manera conjunta hemos realizado las clase **Date**, la interfaz **Taxable** y el informe de la práctica así como el **main** del programa en la clase **OnlineStore**.