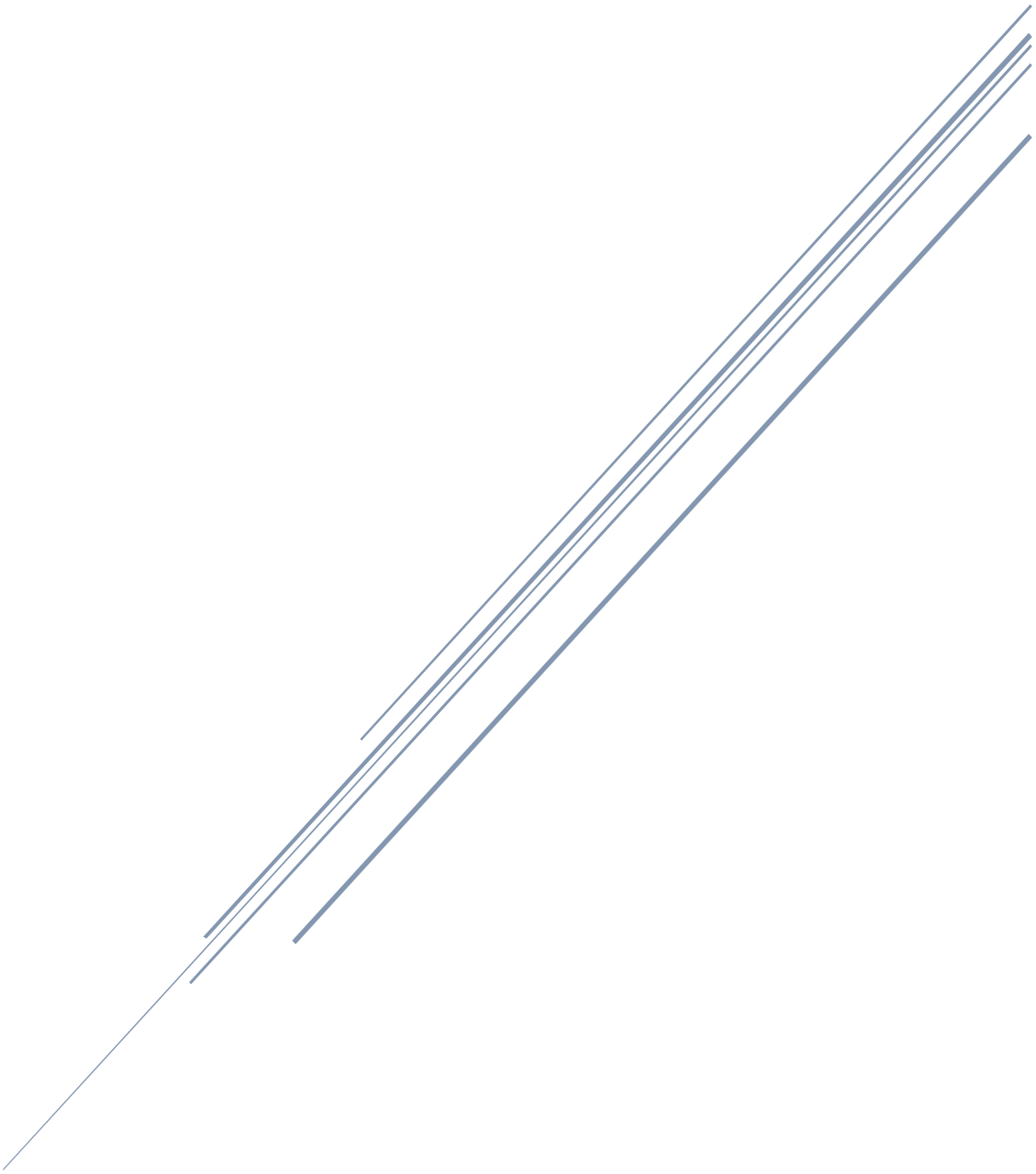
LAB SESSION 4: INTERFACES



Amanda Pintado Lineros u137702
Antonio Pintado Lineros u172771

El objetivo de esta practica consiste en implementar una serie de interfaces, en concreto 2, comparable y taxable, los objetos que implementen taxable deveran de implementar una serie de funciones, para calcular precios en función de un atributo fijo y estatico: tax.

Por otra parte en vez de utilizar fechas como strings como en la anterior practica alguna que otra clase devera de guardar fechas de tipo Date, que es una clase ya creada en Java, que podremos implementar usando import.

Tambien se tendrán que implementar una serie de funciones en nuestra clase onlineStore que nos ayudarán a vender objetos, incrementar la fecha de nuestra onlineStore, controlar audiciones, llevar a cabo sales…

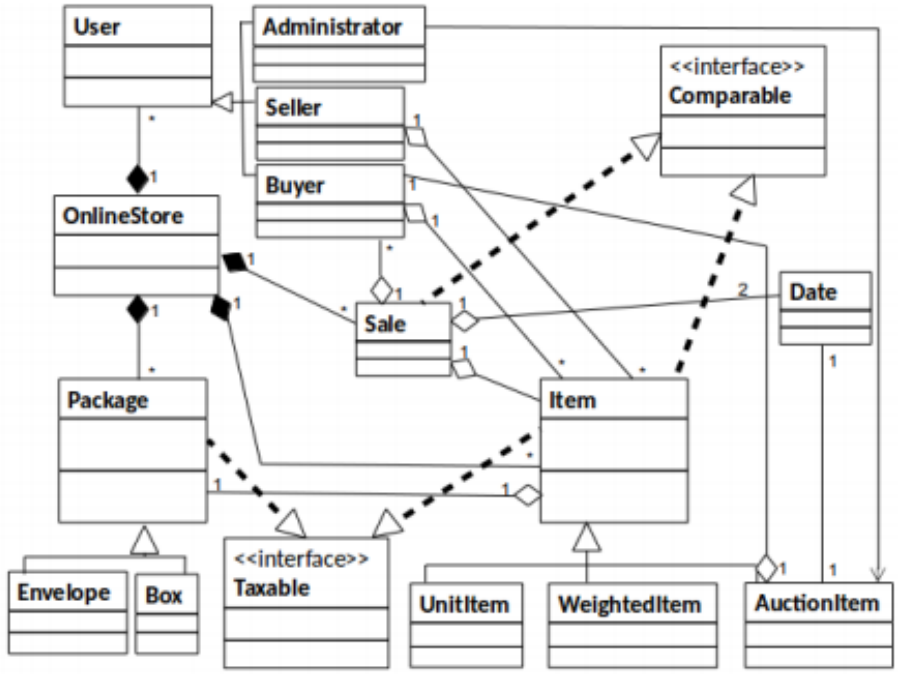


Figure 1: OnlineStore Class Diagram

Primero empezamos creando la interfaz Taxable, esta clase la deberán de implementar las clases Item y Package, a su vez deberán de implementar los métodos getPrice(), getPriceOnlyTax(), getPricePlusTax() y sumTotalTax(Taxable t). O si no los implementan estas, lo deberán de hacer las clases hijas de las mismas, ya que estarían heredando de una clase abstracta, como es en el caso de Item, y su método getPrice(), que quedará definido en sus clases hijas.

Como en la clase Package no quedaba definido en ninguna documentación el precio que debería de retornar al hacer getPrice(), tomamos como precio las medidas del paquete. Podriamos haber optado por una implementación más compleja en función de si fuera un sobre o caja, pero lo decidimos hacer así para que sea mas visual y cumpliera con la implementación que se nos pedía en el laboratorio.

```java
public interface Taxable {

    public static final double tax = 0.21;

    public double getPrice();

    public double getPriceOnlyTax();

    public double getPricePlusTax();

    public double sumTotalTax(Taxable t);

}
```

Implementación de la interfaz Taxable.

```java
@Override
public double getPriceOnlyTax() {
    return this.getPrice() * Item.tax;
}

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

@Override
public double sumTotalTax(Taxable t) {
    return (this.getPriceOnlyTax() + t.getPriceOnlyTax());
}
```

Override de los métodos de la clase implementada Taxable, estas funciones son prácticamente iguales para las clases Item y Package, por otra parte tenemos el método getPrice que ya estaba definido en el laboratorio anterior para la clase Item, pero para el paquete no, por lo que se comportaran de manera diferente.

Una vez hecho esto creamos la clase Sale, esta clase tendrá como atributos una fecha de esta venta, una fecha de envío, un comprador y el Item. A parte en nuestra clase OnlineStore también guardaremos un atributo como LinkedList que guardará todas las Sales.

```
public class Sale implements Comparable {

    private Date saleDate;
    private Date sendDate;
    private final Buyer buyer;
    private final Item item;
```

Esta clase tendrá como metodos unos cuantos getters, getSaleDate, getSendDate, getBuyer y getItem. Dos constructoras, una que será usada cuando querramos especificar la fecha de envio, y otra que no será necesario ya que no se sabe.

Y otros métodos como son setSaleDate y setSendDate.

Por último tenemos el método compareTo, que se comentará más adelante.

Como ya se ha comentado la clase Sale tendra dos atributos de clase Date, pero por otra parte AuctionItem también tendrá un atributo de esta clase, este atributo será deadline, que ha sido cambiado de tipo string a tipo Date, esto conlleva en que teníamos que retocar una serie de funciones, pero que no fue de gran complicación.

Por otra parte, las clases Sale e Item implementarán la interfaz Comparable, por lo que tendrán que hacer un Override del método compareTo, estas dos están implementadas de la siguiente manera: (teniendo en cuenta que queremos ordenar ventas por fecha (saleDate) e Items por precio)

```
@Override
public int compareTo(Object o) {
    //El objeto que entra será un Sale, para hacer
    //la comparación
    Sale s2 = (Sale) o;
    Date d1 = saleDate;
    Date d2 = s2.getSaleDate();
    return d1.compareTo(d2);
}
```

```java
@Override
public int compareTo(Object o){
    Item i2 = (Item) o;
    if(this.getPrice()<i2.getPrice()){
        return -1;
    }
    else if (this.getPrice()==i2.getPrice()){
        return 0;
    }
    else{
        return 1;
    }
}
```

Para comprobar que esto funcione se han realizado una serie de comprobaciones en el main, como es ordenar los ítems vendidos, ordenar los ítems disponibles y por ultimo ordenar las ventas, para ello hemos utilizado el método estatico sort, disponible en la clase Collections:

```
Lista Items Available antes de ser ordenados
1-Mesa, price: 47.0

2-SustratoUniversal, price: 3.0

Lista Items Available despues de ser ordenados
1-SustratoUniversal, price: 3.0

2-Mesa, price: 47.0

Lista Items Sold antes de ser ordenados
1-Sofa, price: 3495.0

2-Libro, price: 105.0

3-Arroz, price: 1.5

4-Auriculares, price: 9.0

5-Ordenador, price: 650.0

Lista Items Sold despues de ser ordenados
1-Arroz, price: 1.5

2-Auriculares, price: 9.0

3-Libro, price: 105.0

4-Ordenador, price: 650.0

5-Sofa, price: 3495.0
```

```
Lista Sales antes de ser ordenadas
1-venta del item Sofa, con fecha 2020/12/16

2-venta del item Libro, con fecha 2020/11/30

3-venta del item Arroz, con fecha 2020/12/01

4-venta del item Auriculares, con fecha 2020/12/03

5-venta del item Ordenador, con fecha 2020/12/06

Lista Sales despues de ser ordenadas
1-venta del item Libro, con fecha 2020/11/30

2-venta del item Arroz, con fecha 2020/12/01

3-venta del item Auriculares, con fecha 2020/12/03

4-venta del item Ordenador, con fecha 2020/12/06

5-venta del item Sofa, con fecha 2020/12/16
```

A más a más, hemos añadido tres métodos a la clase OnlineStore, uno llamado sell, este método recorrerá todos los ítems del array Items Avaiable y los vendera a uno de los compradores, a parte de utilizar el método sell de la clase Seller para que el vendedor especificado como parámetro lo venda. Por otra parte, como se especifica en la práctica también se crearán instancias de la clase Sale en nuestra linkedlist sales para así tenerlas disponibles.

Por último, se retorna una fecha, esta fecha será la general de nuestro main en la cual se guardará la fecha actual de la tienda, ya que se han hecho ventas y para simular que cada venta se realiza un día distinto se ha decidido hacer de esta manera.

```
public static Date sell(Seller aSeller, Date aDate) {
    //Reutilizamos practicamente el mismo código que implementamos en la
    //práctica anterior para simular las ventas
    for (int i = 0; i < itemsAvailable.size(); i++) {
        User actualUser = (User) users.get(i);
        //Todos los usuarios buyer compraran algo
        if (actualUser instanceof Buyer) {
            Buyer actualBuyer = (Buyer) actualUser; //downcast
            Item actualItem = itemsAvailable.get(i);
            //Habra que diferenciar entre Items Unit y Weighted
            if (actualItem instanceof UnitItem) {
                UnitItem actualUnit = (UnitItem) actualItem;
                if (actualUnit.getQuantityRemaining() > 0) {
                    actualBuyer.Buy(actualItem);
                    actualUnit.sell(1);
                    totalPrice += actualItem.getPricePlusTax();
                }
            } else if (actualItem instanceof WeightedItem) {
                WeightedItem actualWeighted = (WeightedItem) actualItem;
                if (actualWeighted.getWeightRemaining() > 0) {
                    actualBuyer.Buy(actualItem);
                    actualWeighted.sell(1);
                    totalPrice += actualItem.getPricePlusTax();
                }
            }
            aSeller.sell(actualItem);

            //Como fecha de envio, como no se espeficifica, aumentaremos en
            //un dia más
```

```
            Calendar c = Calendar.getInstance();
            c.setTime(aDate);
            c.add(Calendar.DATE, 1);
            Date sDate = c.getTime();
            sales.add(new Sale(aDate, sDate, actualBuyer, actualItem));

            totalProfit += actualItem.calculateProfit();
            itemsSold.add(actualItem);
            itemsAvailable.remove(actualItem);
            //Para que cada Sale tenga una fecha de venta distinta,
            //usaremos la variable sDate, que tiene un dia mas que aDate
            //y la usaremos para hacer la nueva venta
            aDate = sDate;
        }
    }
    //Hemos modificado la fecha de la tienda al hacer las ventas, así que
    //habrá que devolverlo
    return aDate;
}
```

Otro método por implementar es el método de incrementDate, este método nos servirá para incrementar la fecha actual de la tienda, a la vez que para comprobar si alguna de nuestras instancias de ítems a subasta expira el día actual, si es así se hará una llamada a la siguiente función con el ítem en cuestión.

```
public static Date incrementDate(Date aDate, LinkedList<AuctionItem> lAuctions, Seller aSeller) {
    //Usamos la clase Calendar para aumentar un dia al Date
    DateFormat dateFormat = new SimpleDateFormat("yyyy/MM/dd");
    Calendar c = Calendar.getInstance();
    c.setTime(aDate);
    c.add(Calendar.DATE, 1);
    aDate = c.getTime();
    String aDateString = dateFormat.format(aDate);
    System.out.println("La fecha a sido cambiada a " + aDateString + "\n");
    for (int i = 0; i < lAuctions.size(); i++) {
        //Compararemos solo el formato simple de fechas
        String auctionDateString = dateFormat.format(lAuctions.get(i).getDeadline());
        if (auctionDateString.equals(aDateString)) {
            System.out.println("Se ha detectado una subasta que ha expirado hoy.\n");
            manageAuction(lAuctions.get(i), aSeller);
        }
    }
    return aDate;
}
```

Como se ha dicho antes, si uno de los objetos a subasta expira el día en el que se hace la compobacion se llamaría a la siguiente función, esta función es manageAuction, esta será la encargada de realizar la venta del objeto al comprador con mayor puja. A parte de actualizar el totalProfit y las listas como es de esperar.

```
public static void manageAuction(AuctionItem aAucIt, Seller aSeller) {
    Buyer actualBuyer = aAucIt.getBuyer();
    Date actualDate = aAucIt.getDeadline();

    actualBuyer.Buy(aAucIt);
    totalPrice += aAucIt.getPricePlusTax();
    aSeller.sell(aAucIt);

    //Como fecha de envio, como no se espeficifica, aumentaremos en
    //un día más
    Calendar c = Calendar.getInstance();
    c.setTime(actualDate);
    c.add(Calendar.DATE, 1);
    Date sDate = c.getTime();
    sales.add(new Sale(actualDate, sDate, actualBuyer, aAucIt));

    totalProfit += aAucIt.calculateProfit();
    itemsSold.add(aAucIt);
    itemsAvailable.remove(aAucIt);
    System.out.println("El objeto " + aAucIt.getName() + " ha sido vendido por el vendedor
}
```

Por último, se han añadido una serie de tests en la clase main, teniendo en cuenta que ahora hay que instanciar métodos de diferente manera, ya que se han añadido atributos Date en unas clases, o se han modificado los parámetros de diversos métodos. Como guía se han añadido comentarios en el código para tener una visión de lo que se esta haciendo en cada punto.

Conclusiones

Como reflexión del laboratorio podemos decir que no hemos tenido grandes complicaciones, ya que la mayoría era reutilizada del laboratorio 3, pero cabe destacar que se ha tenido algún inconveniente a la hora de crear la clase Date, por que primero se ha intentado utilizar una clase propia, que simulara el funcionamiento de Date ya implementada en java, pero no se obtuvo un resultado esperado. Ya que había funciones un tanto mas complejas de lo que esperábamos. Así que se decidió utilizar la clase Date, directamente mediante el import.