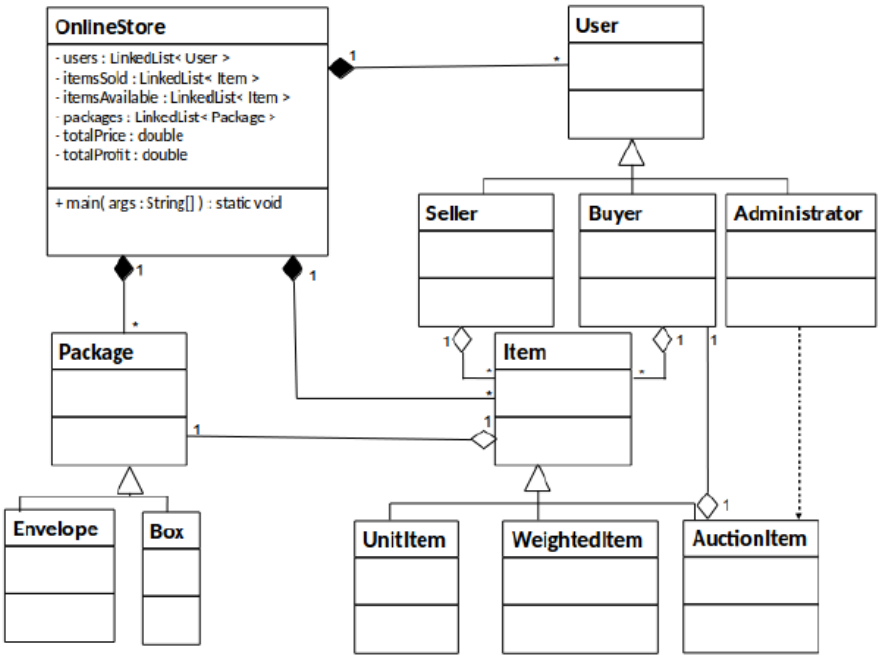
Lab3
Programación Orientada a Objetos

-Descripción del problema a solucionar

En este laboratorio deberemos implementar el diseño realizado en el seminario 3, 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.



Éste sería el diseño del Seminario 3 que nos permite hacernos una idea de como se relacionan las diferentes clases que implementan esta tienda en línea.

Como podemos ver, deberemos implementar un total de 12 clases con sus respectivos atributos y métodos característicos. Muchas de ellas al mantener una relación de herencia con su clase padre, heredaran los métodos y atributos de esta clase, por lo tanto, tendremos que redefinir métodos o simplemente heredarlos. Tendremos que implementar las siguientes clases:

OnlineStore

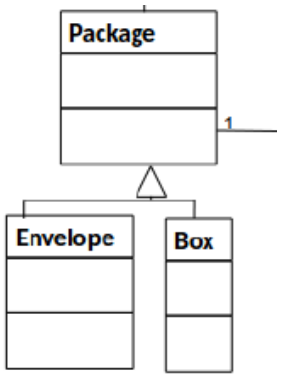
Package → Envelope, Box

Item → UnitItem, WeightedItem, AuctionItem

User → Buyer, Seller, Administrator

-Packaging

En cuanto al **packaging**, deberemos implementar la clase **Package** y las clases que heredan de ésta (**Envelope** y **Box**). Para hacerlo, implementaremos las herencias que se ven en el propio diseño e implementaremos los métodos correspondientes de cada clase:



- Clase Package

Clase padre de los paquetes, en ésta definiremos los atributos y métodos básicos que deberá tener un paquete para poder implementar correctamente nuestra solución.

```java
public class Package {

    private double width;
    private double height;

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


    public double getWidth(){
        return width;
    }


    public double getHeight(){
        return height;
    }


    public void setWidth(double w){
        width = w;
    }


    public void setHeight(double h){
        height = h;
    }
}
```

Como podemos observar, nuestra clase **Package** tendrá dos atributos, **width** y **height** (anchura y altura). También podemos ver el método constructor de la clase y sus **getters** y **setters**.

Se trata de una clase muy sencilla con dos atributos y 5 métodos.

En el constructor deberemos asignar los valores que entran por parámetro del método a los atributos de la clase.

En los **getters** deberemos devolver el atributo de la clase que se nos pide.

En los **setters** deberemos cambiar el valor del atributo por el que entra por parámetro del método.

- Clase Envelope

Esta será la clase correspondiente a los paquetes de tipo embalaje. Este tipo de empaquetado lo recibirán los **items** de la tienda cuya **depth** sea menor que **3 cm**.
Tendremos 3 tipos de embalajes según su tamaño **A3 (29x42), A4 (21x29) y A5 (21x11)**.
En el caso de que el **item** tenga una profundidad menor que 3 pero no quepa en uno de los embalajes se comprobará si cabe en alguna de las cajas de tamaños predefinidos que mostraremos más adelante.

```
public class Envelope extends Package {

    private String name;


    public Envelope(int w, int h, String n){
        super(w, h);
        name = n;
    }


    public String getName(){
        return name;
    }


    public void setName(String n){
        name = n;
    }


    public Boolean isSuitable(double[] size){

        return size[0] <= getWidth() && size[1] <= getHeight() || size[1] <= getWidth() && size[0] <= getHeight();
    }
}
```

Para empezar, como la clase **Envelope** hereda de la clase **Package**, declaramos la clase **Envelope** como la clase que hereda de la clase **Package** con el **keyword extends**.

El único atributo que diferenciará al embalaje un paquete normal será el nombre (**A3, A4, A5**). Por lo tanto, en el método constructor de la clase usaremos **super** para llamar al constructor de la clase padre con los atributos necesarios y después asignaremos el nombre que entra por parámetro del método al atributo **name** del envoltorio.

Vemos que tenemos un **getter** y un **setter** debido a que tenemos un solo atributo de más. Para acceder a los atributos que hereda tenemos los métodos que hereda de la clase padre (**Package**).

Por último, vemos que tenemos un método que nos permite saber si un tamaño determinado se puede encabezar en un envoltorio de los predefinidos anteriormente. Para saberlo comprobaremos que el tamaño del elemento es menor o igual que el tamaño de nuestros envoltorios usando **getWidth**, **getHeight** y **getDepth**.
Para implementar de manera correcta este método se deberá comprobar que el **item** puede entrar de todas las maneras posibles girándolo, por lo tanto, giramos el **size** (giro del **item**) e intentamos meterlo en el envoltorio, devolvemos **true** si cabe y **false** si no cabe.

- Clase Box

Esta será la clase correspondiente a los paquetes de tipo caja, es decir, sera el paquete que utilizarán los **items** de profundidad mayor a **3 cm** o los **items** de profundidad menor que **3 cm** que no caben en los embalajes predeterminados.

Antes de ver el código, debemos hacer mención a que nuestra implementación de la tienda online presenta 6 tipos de cajas predefinidos según sus tamaños:

10x10x10 / 10x10x100 / 10x100x100 / 100x100x100 / 100x150x300 / 200x300x500

Por lo tanto, nuestro tamaño máximo de caja es de **200x300x500** y según nuestra implementación, un **item** que supere este tamaño no tendrá un empaquetado asignado y se deberá contactar con el vendedor del **item** para asignar un paquete (hipotéticamente hablando).

```
public class Box extends Package{

    private double depth;

    public Box(int w, int h, int d){
        super(w,h);
        depth = d;
    }


    public double getDepth(){
        return depth;
    }


    public void setDepth(double d){
        depth = d;
    }
```

Para empezar, vemos que la clase **Box** extiende la clase **Package** dado que hereda de ésta. También podemos ver que el atributo diferencial en este caso no es el nombre sino la **profundidad** dado que será importante para asignar la caja correcta.

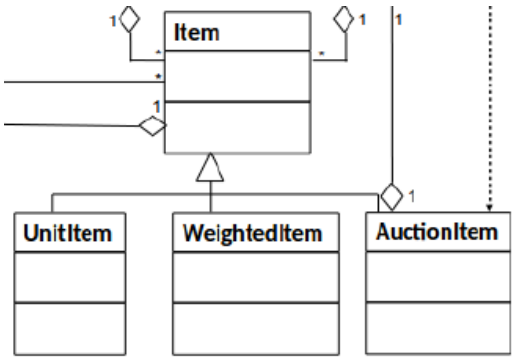
Vemos como se llama al constructor de la clase padre con **super** y se asigna el valor del atributo **depth** en el constructor. Y también vemos como tiene un **getter** y un **setter**, al igual que en la clase anterior dado que tenemos un solo atributo.

Al igual que en el caso de la clase **Envelope**, encontramos el método **isSuitable**, el cuál comprueba si el tamaño del **item** puede entrar en una de las cajas. La diferencia principal entre este método en la clase **Envelope** y en la clase **Box** será que en ésta tenemos en cuenta la profundidad, por lo tanto, deberemos intentar entrar el elemento en la caja de tamaño máximo de todas las maneras posibles (**combinaciones de 3 números**) como vemos:

```
public Boolean isSuitable(double[] size){

    if(size[0] <= getHeight() && size[1] <= getWidth() && size[2]<=getDepth()){

        return true;

    }else if(size[0] <= getHeight() && size[1] <= getDepth() && size[2] <= getWidth()){

        return true;

    }else if(size[0] <= getWidth() && size[1] <= getHeight() && size[2] <= getDepth()){

        return true;

    }else if(size[0] <= getWidth() && size[1] <= getDepth() && size[2] <= getHeight()){

        return true;

    } else if(size[0] <= getDepth() && size[1] <= getWidth() && size[2] <= getHeight()){

        return true;

    }else if(size[0] <= getDepth() && size[1] <= getHeight() && size[2] <= getWidth()){

        return true;
    }
    return false;
}
```

-Item

En cuanto a los items de nuestra tienda online, seguiremos el diseño del seminario por lo que tendremos una clase padre llamada Item y 3 clases hijas llamadas UnitItem, WeightedItem y AuctionItem:



- Clase Item

Ésta será la clase padre de los artículos que se venderán en la tienda online. Cada **item** tendrá un **nombre**, un tipo, un **array** con su tamaño, un **coste** y un **empaquetado**.
También tendrá sus **getters** y **setters** además de un método muy importante para asignar el paquete correspondiente a un **item** según su tamaño. También presentará dos métodos abstractos (**getPrice** y **calculateProfit**) que se implementarán en las clases hijas según el tipo de **item** que estemos tratando.

```
public abstract class Item{

    private String name;
    private String type;
    private double[] size;
    private double cost;
    private Package pack;

    Item(){

        name = "Undefined";
        type = "Undefined";
        size = new double[3];
        for(int i = 0; i<=2; i++){
            size[i] = 0;
        }
        cost = 0;
    }

    Item(String n, String t, double[] s, double c){

        name = n;
        type = t;
        size = new double[3];
        for(int i = 0; i<=2; i++){
            size[i] = s[i];
        }
        cost = c;
    }
```

Como podemos ver, tenemos dos constructores para la clase **Item**, un constructor hace referencia a la creación de un Item del cual no definimos sus atributos y por lo tanto solamente queremos que se cree pero con valores por defecto **indeterminados**.

Seguidamente, podemos ver el constructor que recibe parámetros de la clase Item, en este si que inicializaremos los atributos de la clase Item con los valores que entran por **parámetro** del método constructor de manera que tendremos la instancia de Item inicializada con estos valores.

- getters y setters

```
public String getName(){
    return name;
}

public String getType(){
    return type;
}

public double[] getSize(){
    return size;
}

public double getCost(){
    return cost;
}

public Package getPackage(){
    return pack;
}

public void setName(String n){
    name = n;
}

public void setType(String t){
    type = t;
}

public void setSize(double[] s){
    for(int i = 0; i<=2; i++){
        size[i] = s[i];
    }
}

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

Aquí podemos ver los diferentes **getters** y **setters** que presentará la clase Item, podemos observar los getters referidos a los atributos de la clase (**getName**, **getType**, **getSize**, **getCost** y **getPackage**) y los **setters** correspondientes (**setName**, **setType**, **setSize** y **setCost**).

Un aspecto a tener en cuenta en esta parte es que para el **setSize** no nos bastaría con escribir **size = s**, puesto que de esta manera no se cambiarían correctamente los valores del array en cada índice. Debemos hacer el **setSize** de la manera que se muestra de manera iterativa para poder reemplazar correctamente los valores del **size**.

Seguidamente, mostraremos uno de los métodos más importantes de esta clase, este método será el método **assignBestPackage** que dada una lista de empaquetados posibles, determina el mejor empaquetado para un item determinado.

Este método nos será útil para decidir cual será el empaquetado correspondiente a cada item de nuestra **OnlineStore**.

Se trata de un código de gran tamaño, por lo tanto, hemos decidido explicarlo en las páginas siguientes y en dos partes (asignación de embalaje y asignación de caja).

- assignBestPackage (1a parte)

```
public void assignBestPackage(LinkedList<Package> Lp){

    double depth = size[2];

    boolean solved = false;

    if(depth < 3){

        for(Package p: Lp){

            if(p instanceof Envelope){

                if(((Envelope) p).isSuitable(size)){

                    pack = p;

                    ((Envelope)pack).setName(((Envelope) p).getName());
                    ((Envelope)pack).setWidth(((Envelope) p).getWidth());
                    ((Envelope)pack).setHeight(((Envelope) p).getHeight());
                    solved = true;
                }
            }
        }
        if(solved){
            System.out.println("Envelope " + ((Envelope)pack).getName() + " assigned to item " + name + ".");
        }
```

En esta primera parte se comprueba si la profundidad del item es **menor que 3 cm**, en el caso de que la profundidad sea menor que esa cifra deberemos buscar un embalaje adecuado para nuestro **item**, como se puede dar el caso de que el item tenga una profundidad menor que 3 cm pero que tenga otras medidas mayores que las de los embalajes disponibles, inicializamos un booleano **solved** en falso que nos indicará en todo momento si hemos solucionado el problema.

Con el booleano **solved** podemos gestionar que si no encuentra un embalaje adecuado para el item lo pueda empaquetar con una caja aunque su profundidad sea menor que 3.

Por lo tanto, comprobamos que la profundidad sea menor que 3, en ese caso iteraremos por la lista de empaquetados disponibles y solo nos interesará comprobar los empaquetados de tipo embalaje, por lo que usaremos **if(p instanceof Envelope)**, esto nos permitirá saber si el empaquetado que en el que nos encontramos de la lista de empaquetados es una instancia de **Envelope**. En el caso de que lo sea, aplicaremos el método **isSuitable** de la clase Envelope, por lo que deberemos realizar un **downcast** a la variable **p** que esta iterando por la lista de empaquetados y usar ese método que hemos explicado anteriormente.

En el caso en que isSuitable devuelva **true** será el caso en el que hemos encontrado el embalaje adecuado para nuestro item, por lo tanto, asignamos al atributo pack el empaquetado en el que nos encontrábamos, y también asignamos al empaquetado su nombre y medida.

Una vez solucionado nuestro booleano solved se cambiará a true y habremos resuelto el problema.

Como hemos mencionado, si no encontrara un embalaje adecuado debido a que las medidas del item son mayores o cualquier otro caso, comprobaría si se puede empaquetar con una caja con las condiciones que veremos a continuación.

- assignBestPackage (2a parte)

```
}if(!solved){

    for(Package p: Lp){

        if(p instanceof Box){

            if(((Box) p).isSuitable(size)){

                Box box = (Box)p;
                double boxwidth = box.getWidth();
                double boxheight = box.getHeight();
                double boxdepth = box.getDepth();

                pack = p;

                ((Box)pack).setWidth(boxwidth);
                ((Box)pack).setHeight(boxheight);
                ((Box)pack).setDepth(boxdepth);
                solved = true;
                break;
            }
        }
    }
    if(solved){
        System.out.println("Box " + "with size {"+ ((Box)pack).getWidth() + ", " + ((Box)pa

    }else{
        System.out.println("No package available for item "+ name + ".");
    }
}
```

En esta segunda parte comprobaremos si el item puede enviarse en una caja, para ello, primeramente, deberemos comprobar que el problema no esta resuelto con el booleano **solved**, en ese caso intentaremos asignar una caja a nuestro item, para ello, como anteriormente, iteraremos por la lista de empaquetados disponibles y comprobaremos que el empaquetado en el que nos encontramos es una instancia de empaquetado tipo caja (**Box**). En el caso de que sea así, comprobaremos si nuestro **item** cabe en la caja en la que nos encontramos de la iteración con **isSuitable** y el **downcast** idénticamente a como lo hemos hecho en la primera parte de explicación de este método.

Si el elemento cabe en la caja asignamos al atributo **pack** del item el paquete en el que nos encontrábamos y el tamaño de este, cambiamos el booleano a true puesto que hemos resuelto el problema y realizamos un **break** debido a que no queremos seguir comprobando. Si siguiéramos comprobando, asignaría la caja de tamaño mayor puesto que en nuestra lista de empaquetados tenemos ordenadas las cajas según su tamaño de menor a mayor.

Como podemos ver, una vez hemos resuelto se imprime una frase diciendo que paquete se le ha asignado al elemento, en el caso en el que el problema no se haya solucionado **imprimiremos por pantalla** que no hay empaquetado disponible para el **item** sobre el que estamos trabajando.

- getPrice y calculateProfit

Estos dos métodos serán métodos abstractos, por lo tanto, se redefinirán en las clases hijas dado que tendrán comportamientos diferentes según el tipo de item con el que estemos tratando.

```
public abstract double getPrice();

public abstract double calculateProfit();
```

- Clase UnitItem

Esta será una de las clases que heredará de la clase Item, esta clase hará referencia los items de tipo unidad en los que tendremos un precio por unidad, una cantidad total y una cantidad restante:

```
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){

        super(n,t,s,c);

        unitPrice = uprice;
        quantity = q;
        quantityRemaining = q;
    }

    public double getQuantityRemaining(){

        return quantityRemaining;
    }

    @Override
    public double getPrice(){

        return unitPrice*quantityRemaining;
    }

    @Override
        public double calculateProfit(){

        return (quantity-quantityRemaining)*(unitPrice-getCost());
    }

    public double sell(int q){

        quantityRemaining = 0;

        return unitPrice*quantity;
    }
}
```

Como podemos ver, tenemos un método constructor que toma los elementos que le entran por parámetro del método y los asigna a los atributos de la clase, también llama al constructor de la clase padre (**Item**) con **super** y los elementos necesarios para entrar por parámetro de ese constructor.

Tendremos dos **getters**, uno nos servirá para tener la cantidad restante en cualquier momento (**getQuantityRemaining**), y otro nos servirá para saber el precio del item (**getPrice**), como podemos ver, **getPrice** se calcula multiplicando el precio de una unidad por la cantidad restante de unidades de ese item.

Seguidamente, podemos ver el método **calculateProfit**, este método nos permite calcular el beneficio de un item. Lo implementamos de manera que devuelve el resultado de la multiplicación de la resta de la cantidad total menos la remanente con la resta de el precio de la unidad con el coste.

Por último, el método sell lo que hará será asignar la cantidad restante a **0** y devolver el resultado de multiplicar el precio de la unidad por la cantidad total.

- Clase WeightedItem

Esta también será una de las clases que heredará de la clase Item, esta clase hará referencia los items de tipo peso en los que tendremos un precio por peso, un peso total y un peso restante:

```java
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){

        super(n, t, s, c);

        pricePerWeight = wprice;
        weight = w;
        weightRemaining = w;
    }

    public double getWeightRemaining(){

        return weightRemaining;
    }

    @Override
    public double getPrice(){

        return pricePerWeight*weightRemaining;
    }


    @Override
    public double calculateProfit(){

        return (weight-weightRemaining)*(pricePerWeight-getCost());
    }


    public double sell(double w){

        weightRemaining = 0;

        return pricePerWeight*weight;
    }
}
```

Como podemos ver, tenemos un método constructor que toma los elementos que le entran por parámetro del método y los asigna a los atributos de la clase, también llama al constructor de la clase padre (**Item**) con **super** y los elementos necesarios para entrar por parámetro de ese constructor.

Tendremos dos **getters**, uno nos servirá para tener el peso restante en cualquier momento (**getWeightRemaining**), y otro nos servirá para saber el precio del item (**getPrice**), como podemos ver, **getPrice** se calcula multiplicando el precio por peso por el peso restante de ese item.

Seguidamente, podemos ver el método **calculateProfit**, este método nos permite calcular el beneficio de un item. Lo implementamos de manera que devuelve el resultado de la multiplicación de la resta del peso total menos el remanente con la resta de el precio por peso con el coste.

Por último, el método sell lo que hará será asignar el peso restante a **0** y devolver el resultado de multiplicar el precio del peso por el peso total.

- Clase AuctionItem

Esta será también una de las clases hijas de la clase Item y hará referencia a los items de tipo **subasta**, estos **items** tendrán un precio que variará según las **apuestas** que se hagan, una fecha limite para pujar, un comprador que también podrá cambiar con el paso del tiempo y por último un impuesto de **5 euros** de la tienda online y un porcentaje del **0,05** para la tienda online también.

```
public class AuctionItem extends Item{

    private double currentPrice;
    private Buyer bidder;
    private String 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, String d){

        super(n,t,s,c);

        currentPrice = startingPrice;
        deadline = d;
    }

    @Override
    public double getPrice(){

        return currentPrice;
    }

    @Override
    public double calculateProfit(){

        if(bidder == null ){

            return fee;
        }
        return fee + getPrice()*percent;
    }

    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()
        }
    }
```

Como podemos ver tenemos los atributos que acabamos de mencionar declarados, un método constructor que funciona igual que los métodos vistos en los otros tipos de items y algunos métodos adicionales.

Para empezar, tenemos un **getter** (**getPrice**) que nos devolverá el precio actual de nuestro item de subasta. Seguidamente, tendremos el método **calculateProfit** que devolverá los 5 euros correspondientes al **fee** en el caso de que nadie haya pujado (**bidder == null**) y los 5 euros más el precio del item por el porcentaje en caso contrario.

También podemos ver el método **makeBid**, este método nos permitirá gestionar que un usuario realice una puja por el **item**, lo implementamos de manera que si la puja es mayor que el precio actual del item, se asigna el **nuevo comprador** como el usuario que ha pujado y el nuevo precio como el **precio** con el que ha pujado, también se imprime por pantalla la acción de la puja y se pide a los demás usuarios que suban su **puja**.

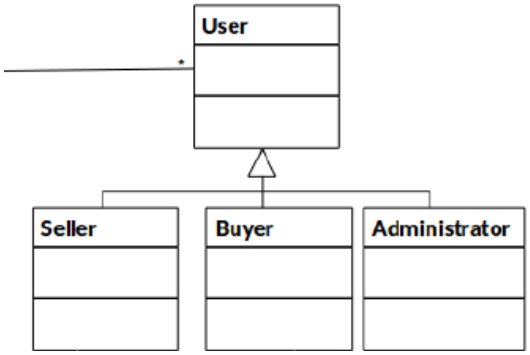
```
public Boolean frozen(String d){

    int actualday = Integer.parseInt(d.substring(0, 2));
    int actualmonth = Integer.parseInt(d.substring(2, 4));
    int actualyear = Integer.parseInt(d.substring(4, 8));

    int lastday = Integer.parseInt(deadline.substring(0, 2));
    int lastmonth = Integer.parseInt(deadline.substring(2, 4));
    int lastyear = Integer.parseInt(deadline.substring(4, 8));

    if(actualyear == lastyear){
        if(actualmonth == lastmonth){
            if(actualday <= lastday){
                return false;
            }
        }else if(actualmonth<lastmonth){
            return false;
        }
    }else if (actualyear<lastyear){
        return false;
    }
    return true;
}

public Buyer getBuyer(){

    return bidder;
}

public String getDeadline(){
    return deadline;
}
```

Por último pero no menos importante, encontramos el método **frozen** y los **getters** de los atributos **Buyer** (comprador) y **Deadline** (fecha límite de la subasta).

Si nos fijamos en el método frozen, podemos ver como hemos desarrollado un algoritmo que dada la fecha de tipo String que entra por parámetro del método, comprueba si la subasta esta acabada en esa fecha, de manera que devuelve false si aun hay tiempo para pujar y la fecha es anterior o igual a la fecha límite, y devuelve true si hemos sobrepasado la fecha límite para pujar por el item y por lo tanto la subasta se ha “congelado” y no se puede pujar.

- User

Esta será la clase que nos permitirá manejar los usuarios de nuestra tienda online. Nuestros usuarios deberán tener nombre, identificador y contraseña en el caso del usuario básico. Si se trata de un **Buyer** o un **Seller** deberán tener también un número de cuenta ascociado a su usuario. Como en las clases vistas anteriormente, seguiremos el diseño del seminario:



- Clase User

Esta será la clase padre de los tipos de usuario, en éste definiremos los atributos y los métodos básicos que deberá tener un usuario de la tienda online:

```java
public class User {

    private String name;
    private final String identifier;
    private final String password;

    public User(String n, String id, String pass){

        name = n;
        identifier = id;
        password = pass;
    }


    public String getName(){
        return name;
    }


    public String getId(){
        return identifier;
    }


    public String getPassword(){
        return password;
    }


    public void setName(String n){
        name = n;
    }


    public Boolean login(String p){
        return password.equals(p);
    }
}
```

Como podemos ver, un usuario tiene un **nombre**, un **identificador** y una **contraseña** como atributos.

Para implementar su método **constructor** deberemos asignar los parámetros del método a los atributos de la clase.

Vemos también que presenta **getters** para cada atributo que nos permitirán acceder a los atributos del usuario en cualquier momento y solo presenta un **setter** para el nombre del usuario dado que será el único atributo que se podrá modificar.

Podemos ver que tendrá un método **login** que le permitirá entrar a la tienda online. Básicamente este método comprobará que la contraseña entrada por parámetro es la misma que la contraseña del usuario, si es la misma devuelve true y si no es la misma devuelve false.

- Clase Buyer

Ésta será la clase correspondiente a los usuarios que compran en la tienda online, esta clase heredará de la clase User y añadirá un atributo para el número de cuenta del usuario y una lista de elementos comprados:

```
public class Buyer extends User{

    private String accountNumber;
    private LinkedList<Item> boughtItems;

    public Buyer(String n, String id, String p, String a){

        super(n, id, p);
        accountNumber = a;
        boughtItems = new LinkedList<Item>();
    }

    public void buy(Item i){

        if(pay(i.getPrice())){

            boughtItems.add(i);

            System.out.println( getName() +" is buying item "+ i.getName() + " for "+ i.getPr
            System.out.println("Price "+ i.getPrice() +" is getting charged into account " +
        }
    }


    private Boolean pay(double price){

        return price>0;
    }
}
```

Como podemos ver, el constructor llama al constructor de la clase padre (**User**) con los atributos correspondientes y seguidamente asigna los parámetros del método a los atributos correspondientes.

Podemos observar el método buy de la clase, en este método primero comprobaremos que el usuario puede pagar (este método podría trabajar-se más creando una cuenta bancaria y una cantidad de dinero), el método **pay** tal como lo hemos planteado devolverá true siempre que el precio sea mayor que **0**, obviamente siempre va a dar true pero como no tenemos una cuenta bancaria con una cantidad de dinero en los usuarios no podemos comprobar que disponga del dinero para pagar.

Una vez hemos comprobado que “puede pagar” con el método pay, añadimos el item comprado a la lista de items comprados del Buyer e imprimimos por pantalla la acción realizada (comprar el item y el precio que nos cuesta).

- Clase Seller

Esta será la clase correspondiente a los usuarios vendedores de la tienda online. Esta clase heredará también de la clase **User** y añadirá un atributo para el número de cuenta, una lista de **items** disponibles y una lista de **items** vendidos:

```
public class Seller extends User {

    private String accountNumber;
    private LinkedList<Item> soldItems;
    private LinkedList<Item> availableItems ;

    public Seller(String n,String id, String p, String a){

        super(n, id, p);
        accountNumber = a;
        soldItems = new LinkedList<Item>();
        availableItems = new LinkedList<Item>();
    }

    public void sell(Item i){

        boolean isthere = false;
        for(Item x: availableItems){
            if(x == i){
                isthere = true;
            }
        }
        if(isthere){
            soldItems.add(i);
            System.out.println( getName() +  " has sold item "+ i.getName() + " and "
        }
    }

    public void addAvailableItem(Item i){
        availableItems.add(i);
    }

    private Boolean deposit(double price){
        return price > 0;
    }
}
```

Como podemos ver, el constructor llama al constructor de la clase padre (**User**) con los atributos correspondientes y seguidamente asigna los parámetros del método a los atributos correspondientes.

Vemos que tenemos un método *sell* el cual primeramente comprobará que el item que queremos vender esta en la lista de items disponibles del **Seller**. En el caso de que lo encontremos lo añadimos a la lista de items vendidos e imprimimos por pantalla la acción realizada (vender el item y el beneficio de este).

También observamos dos métodos **addAvailableItem** y deposit, el primero nos añade el item que entra por parámetro del método a la lista de items disponibles del **Seller** y el segundo funciona como el método pay de la clase **Buyer**.

Podríamos mejorar los métodos **pay** y **deposit** añadiendo una cantidad de dinero en cada usuario que cambiaría en función de las ventas y las compras.

- Clase Administrator

Esta será la clase que hará referencia a los usuarios administradores, al igual que las anteriores, también heredará de la clase User. Estos usuarios serán los encargados de administrar las subastas de artículos:

```
public class Administrator extends User{

    public Administrator(String n, String id, String p){
        super(n, id, p);
    }

    public Boolean expel(User u){

        System.out.println(getName() + " has expelled the user " + u.getName() + ".");
        return true;
    }

    public Boolean manageAuction(AuctionItem a, String date){

        if(!a.frozen(date)){
            System.out.println(getName() + " is managing the item " + a.getName() + "
            return true;
        }
        return false;
    }

    public void printStock(LinkedList<AuctionItem> item){

        System.out.println("The administrator "+ getName() + " is going to show the au

        for (Item i: item){
            System.out.println(i.getName() + " has current price of " + i.getPrice()+
        }
    }
}
```

Como podemos ver, el constructor llama al constructor de la clase padre (**User**) con los atributos correspondientes y seguidamente asigna los parámetros del método a los atributos correspondientes.

En cuanto al Administrador podemos observar diferentes métodos, para empezar, podemos ver el método **expel**, este método lo hemos planteado de manera abstracta puesto que no tenemos acceso desde la clase **Administrator** a la **lista de usuarios**, por lo tanto, simplemente imprimimos por pantalla que eliminamos al usuario, devolvemos true y en el **main** de nuestra **tienda online** lo eliminaremos de la lista de usuarios.

Seguidamente, podemos ver el método **manageAuction**, en este método primero comprobamos que la fecha en la que nos encontramos sea una fecha anterior o igual a la fecha limite con el método **frozen**. En el caso de que la subasta no este finalizada imprimimos por pantalla que el administrador esta manejando la subasta y devolvemos **true**, en caso contrario devolvemos **false**.

Por último, vemos el método **printStock**, este método imprimirá por pantalla el **stock** de **items en subasta** y quién los esta mostrando (el administrador).

- Clase OnlineStore

Esta será la clase en la que realizaremos las acciones de la tienda online para comprobar que todo funciona correctamente. Lo que haremos será crear usuarios, **items** y paquetes, y simularemos acciones que se podrían llevar a cabo por los usuarios como: comprar, vender, realizar una subasta, pujar y demás. Finalmente, se imprimirán todas las acciones de los usuarios de la tienda y el Precio y Beneficio total de la tienda online.

```
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 double totalPrice;
    public static double totalProfit;

    public static void init(){

        itemsAvailable = new LinkedList< Item >();
        itemsSold = new LinkedList< Item >();
        users = new LinkedList< User >();
        packages = new LinkedList< Package >();
        totalPrice = 0.0;
        totalProfit = 0.0;
    }
```

Como podemos ver, la clase **OnlineStore** tendrá como atributos 4 listas para almacenar los **artículos disponibles**, los **artículos vendidos**, los **usuarios** y los **empaquetados disponibles**. También tendrá un atributo **totalPrice** y un atributo **totalProfit** que mantendrán el precio total de la tienda y el **beneficio** total de ésta.

Para inicializar los atributos creamos un método init en el que los inicializamos que usaremos más adelante.

```
public static void main(String[] args) {
    // TODO code application logic here
    init();

    //AÑADIMOS ITEMS
    itemsAvailable.add(new UnitItem("Gaming Chair", "Furniture", new double[]{53.0, 110.0, 160.0}, 300, 500, 2));
    itemsAvailable.add(new WeightedItem("Rice", "Food", new double[]{12.0, 22.0, 2.0}, 1.5, 2.5, 50));
    itemsAvailable.add(new UnitItem("TV", "Appliance", new double[]{100.0, 60.0, 10.0}, 600, 1000, 4));

    //AÑADIMOS COMPRADORES VENDEDORES Y UN ADMINISTRADOR
    users.add(new Buyer("Marcos Navajas", "Markito", "aylutequiero", "12345678"));
    users.add(new Buyer("Oscar Bujía", "Oscaaaaaaaar", "pollo", "123456789"));
    users.add(new Buyer("Didac Sitjas", "DidacTempo", "6969", "12345678910"));
    users.add(new Seller("Daniel Leorri", "Wade", "elbaileesmivida67", "1234567891011"));
    users.add(new Administrator("Albert Gubau", "Albert", "contraseña"));

    //AÑADIMOS LOS PAQUETES DISPONIBLES
    packages.add(new Envelope(29, 42, "A3"));
    packages.add(new Envelope(21, 29, "A4"));
    packages.add(new Envelope(21, 11, "A5"));
    packages.add(new Box(10, 10, 10));
    packages.add(new Box(10, 10, 100));
    packages.add(new Box(10, 100, 100));
    packages.add(new Box(100, 100, 100));
    packages.add(new Box(100, 150, 300));
    packages.add(new Box(200, 300, 500));
```

Seguidamente, creamos items, usuarios y empaquetados de todos los tipos y los añadimos a las listas anteriormente creadas e inicializadas con el método **init**.

```
System.out.println("PACKAGE ASSIGNMENT FOR THE ITEMS AVAILABLE:\n");

//ASIGNAMOS EL MEJOR PAQUETE PARA CADA ITEM
for(int i = 0; i<itemsAvailable.size(); i++){
    itemsAvailable.get(i).assignBestPackage(packages);
}

//TOMAMOS EL TERCER USUARIO COMO UN SELLER
Seller seller = (Seller)users.get(3);

//LLENAMOS LA LISTA DEL SELLER CON LOS ITEMS QUE HEMOS AÑADIDO
for(int i = 0; i<itemsAvailable.size(); i++){
    seller.addAvailableItem(itemsAvailable.get(i));
}
```

Después asignamos los empaquetados correspondientes a todos los **items** creados con el método anteriormente explicado **assignBestPackage**.

Asignamos el rol de **seller** al tercer usuario de la lista de usuarios y le añadimos a la lista de items disponibles del seller los **items** disponibles de la **tienda online**.

```
System.out.println("\nUSERS SHOPPING:\n");

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

    Item item = itemsAvailable.get(i);
    Buyer b = (Buyer)users.get(i);

    b.buy(item);

    totalPrice += item.getPrice();

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

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

    seller.sell(item);

    totalProfit += item.calculateProfit();

    itemsSold.add(item);
}


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

    itemsAvailable.remove(i);
}
```

Seguidamente, simulamos a los usuarios comprando de manera que iteramos por la lista de items disponibles asignando un comprador a cada **item**, aplicamos el método **buy** del **Buyer**, aumentamos el precio total de la tienda y aplicamos el método **sell** según el tipo de instancia del **item** que compramos.

En este caso hemos implementado la tienda de manera que cada vez que se compra un item se compra toda la cantidad de este.

Después aplicamos el método **sell** del **Seller** para poder ampliar la lista de items vendidos del **seller**. Aumentamos los beneficios totales mediante el método **calculateProfit** y añadimos el item a la lista de items vendidos.

Por último, **borramos los items** vendidos de la lista de **items disponibles** de manera iterativa.

```
System.out.println("\nCOMIENZO DE LA PUJA:\n");

//COMIENZO DE LA PUJA
LinkedList<AuctionItem> lai = new LinkedList<AuctionItem>();
Administrator admin = (Administrator)users.get(4);
AuctionItem auctionitem = new AuctionItem("Armario", "Furniture",
lai.add(auctionitem);

System.out.println("PACKAGE ASSIGNMENT FOR THE AUCTION ITEM:\n");

auctionitem.assignBestPackage(packages);
seller.addAvailableItem(auctionitem);
itemsAvailable.add(auctionitem);

System.out.println("\nPRESENTATION AND START OF THE AUCTION:\n");

admin.printStock(lai);
```

En la parte siguiente comenzamos la simulación de la puja, por lo tanto, creamos una lista de items para subasta, asignamos un usuario como **Administrador**, creamos un nuevo item de tipo **AuctionItem** y lo añadimos a la lista creada de items para subasta.

Después, asignamos el empaquetado al **item** de subasta con el método **assignBestPackage** explicado anteriormente, añadimos el item de subasta a la lista de items disponibles y hacemos que el administrador imprima por pantalla el **stock** de items de subasta con **printStock**.

```
System.out.println("\nSTART OF THE AUCTION:\n");

if(!auctionitem.frozen("06112020")){
    auctionitem.makeBid((Buyer)users.get(1), 10500.0);
}

if(!auctionitem.frozen("07112020")){
    auctionitem.makeBid((Buyer)users.get(0), 11000.0);
}

if(!auctionitem.frozen("10112020")){
    auctionitem.makeBid((Buyer)users.get(2), 12000.0);
}

admin.manageAuction(auctionitem, "11112020");

if(!auctionitem.frozen("11112020")){
    auctionitem.makeBid((Buyer)users.get(1), 13000.0);
}

admin.expel(users.get(0));
users.remove(users.get(0));

Buyer buyer = auctionitem.getBuyer();
buyer.buy(auctionitem);
seller.sell(auctionitem);
```

En esta parte simulamos las pujas de los usuarios de la tienda online de manera que comprobamos si en la fecha en la que nos encontramos la subasta ya se ha terminado con el método **frozen**, en el caso de que la subasta no se haya terminado pujamos una cantidad adecuada para el precio del **item**. Aplicamos el método **manageAuction** un día de la **finalización** de la subasta y seguidamente expulsamos al primer usuario. Finalmente, tomamos el **comprador final** del artículo y aplicamos el **buy** y el **sell** de **Buyer** y **Seller** correspondientemente.

```
totalPrice += auctionitem.getPrice();
totalProfit += auctionitem.calculateProfit();

System.out.println("Total price: " + totalPrice);
System.out.println("Total profit: " + totalProfit);
```

Para terminar, aumentamos el precio de la tienda con el precio del **item de subasta** y el beneficio con el beneficio obtenido por el **item de subasta**, y también imprimimos por pantalla el **precio total** y el **beneficio total**.

- Conclusiones, Soluciones Alternativas y Contribuciones

En conclusión de la práctica, nos ha parecido una práctica con demasiadas clases y que no ha estado diseñada de la mejor manera, hemos tenido problemas hasta el último momento debido a la multiplicidad de clases que se nos han presentado con métodos que a veces eran meramente hipotéticos como el método **pay** dado que no disponemos de una cantidad de dinero como atributo de los usuarios.

Una solución alternativa sería tener como atributo de cada usuario el dinero que tienen en la cuenta bancaria de manera que sería más sencillo e intuitivo comprar y vender artículos.
También un cambio que se podría realizar es que el usuario decidiera que cantidad de items quiere comprar de la cantidad total de items disponibles en vez de comprar toda la cantidad disponible cuando compra un **item**. Este cambio se podría realizar añadiendo un parámetro cantidad al método **buy** y al método **sell** de manera que se tuviera en cuente que cantidad o peso del elemento quieres comprar. Además de este cambio, al asignar un paquete a un item no solo se debería de comprobar si la profundidad es menor que 3 cm sino que se debería comprobar si una de todas las medidas del tamaño del item es menor que **3 cm**, puesto que, en ese caso deberá tener un **envoltorio** como empaquetado.

En el trabajo total realizado, **Oriol Estabanell** ha realizado las partes de los usuarios (**Buyer, Seller y Administrator**), la parte de la simulación de las compras en la clase **OnlineStore**, parte del método **assignBestPackage** de los **items** y parte de las **soluciones alternativas** para la práctica.

Albert Gubau ha realizado la parte de los **items** (**Unititem, WeightedItem y AuctionItem**) y el empaquetado de estos y gran parte de la clase **OnlineStore**.