INTRODUCCIÓN

El objetivo de esta práctica era implementar el diseño del Seminario 5 que modela una librería online. El propósito principal ha sido implementar las cuatro clases Stock, Book, Catalog, ShoppinCart y el main, TestStore, para testear el programa, donde ejecuta la aplicación de tipo Libreria. En resumen, nuestro deber ha residido en decidir cómo cada método debería funcionar en vez de programarlo, y asegurarnos de que funcionaban correctamente en todos los casos las clases del programa. En este report, hemos explicado brevemente la implementación de estas clases y métodos que hemos utilizado y hemos dejado de utilizar para hacer el código más limpio y entendible. Hemos finalizado el report con una recopilación de todos los detalles de cómo nos hemos organizado en esta práctica.

EXPLICANDO EL CÓDIGO

CLASE STOCK

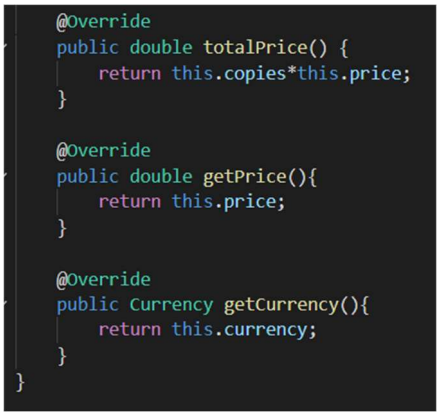
```
package bookstore;
import java.util.Currency;
public class Stock implements StockInterface {
    protected Book book;
    protected int copies;
    protected double price;
    protected Currency currency;
    public Stock(Book bookinit, int copinit, double priceinit, Currency curinit) {
        this.book = bookinit;
        this.copies = copinit;
        this.price = priceinit;
        this.currency = curinit;
    }
    public Book getBook() {
        return book;
    }
    @Override
    public String getBooktitle() {
        return book.getTitle();
    }

    @Override
    public int numberOfCopies() {
        return this.copies;
    }

    @Override
    public void addCopies(int numberOfCopies) {
        this.copies += numberOfCopies;
    }

    @Override
    public void removeCopies(int numberOfCopies) {
        this.copies -= numberOfCopies;

    }
```



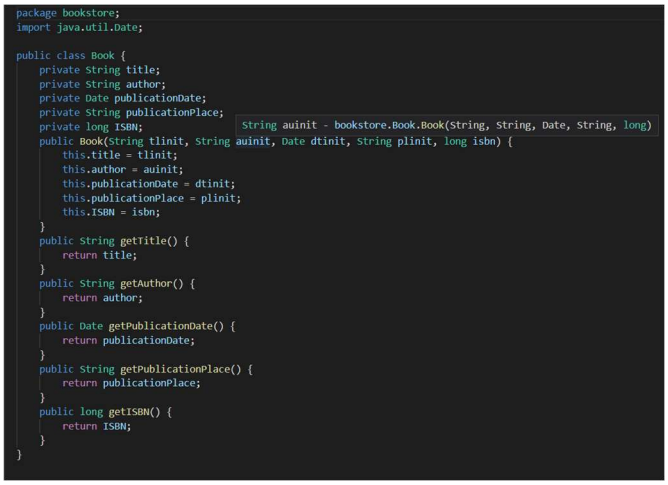
```
    @Override
    public double totalPrice() {
        return this.copies*this.price;
    }

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

    @Override
    public Currency getCurrency(){
        return this.currency;
    }
}
```

Hemos creado la *clase Stock* para gestionar el stock disponible de cada libro. Es decir, nuestro programa cuando lo ejecutamos empieza con x copias de cada libro y una vez que vamos comprando, esas copias van disminuyendo. En esta clase está incluida, *implements*, en la declaración para especificar una interfaz llamada StockInterface. Esta clase implementa los tipos y métodos abstractos declarados por las interfaces y se deben sobre escribir. Además, está compuesta por cuatro atributos, **book, copies, Price** y **currency** que en este caso son *protected*, y solo pueden ser accedidos por miembros de su propia clase, de subclases de esa clase o clases del mismo paquete. En el apartado de los métodos nos encontramos con el constructor **Stock**, que inicializa todos los atributos declarados anteriormente. Además, tenemos un getter **getBook()**, para obtener el libro que nos han pedido. Por otro lado, hemos sobre escrito los métodos implementados desde StockInterface. Han sido los siguientes métodos los que hemos sobre escrito: **getBookTitle()**, para obtener el título del libro, **numberOfCopies(),** para saber cuántas copias nos queda de stock, **addCopies()**, añade copias, **removeCopies()**, elimina copias del "almacen" que tenemos en la librería online, **totalPrice()**, devuelve el precio total que nos van a costar las copias que hemos comprado, **getPrice()**, obtiene el precio del libro, **getCurrency()**, obtiene la divisa en la que está puesta el precio del libro.

CLASE BOOK

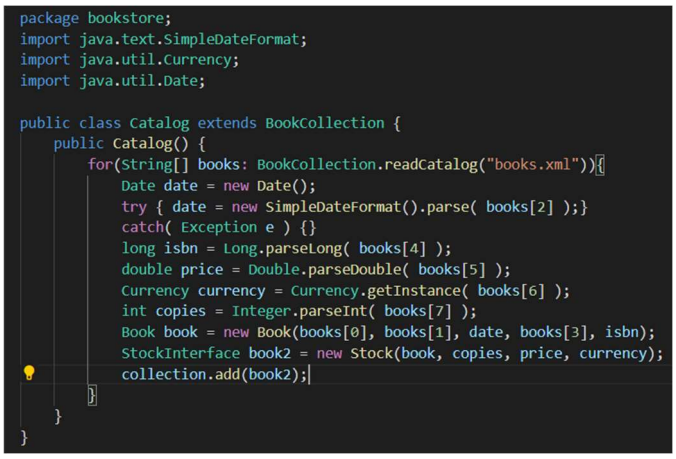


```java
package bookstore;
import java.util.Date;

public class Book {
    private String title;
    private String author;
    private Date publicationDate;
    private String publicationPlace;
    private long ISBN;
    public Book(String tlinit, String auinit, Date dtinit, String plinit, long isbn) {
        this.title = tlinit;
        this.author = auinit;
        this.publicationDate = dtinit;
        this.publicationPlace = plinit;
        this.ISBN = isbn;
    }
    public String getTitle() {
        return title;
    }
    public String getAuthor() {
        return author;
    }
    public Date getPublicationDate() {
        return publicationDate;
    }
    public String getPublicationPlace() {
        return publicationPlace;
    }
    public long getISBN() {
        return ISBN;
    }
}
```

Esta clase **book** está compuesta por unos atributos que son los siguientes: **title**, **author, publicationDate, publicationPlace, ISBN.** Además, tenemos un constructor llamado **Book**, para inicializar todos los atributos declarados anteriormente. Por otro lado, también hemos utilizado **getters, getTitle()**, para obtener el título, **getAuthor(),** para obtener el autor del libro, **getPublicationDate(),** para obtener la fecha de publicación, **getPublicationPlace(),** para obtener el lugar de publicación y finalmente **get ISBN,** para obtener el número ISBN del libro.

CLASE CATALOG



```java
package bookstore;
import java.text.SimpleDateFormat;
import java.util.Currency;
import java.util.Date;

public class Catalog extends BookCollection {
    public Catalog() {
        for(String[] books: BookCollection.readCatalog("books.xml")){
            Date date = new Date();
            try { date = new SimpleDateFormat().parse( books[2] );}
            catch( Exception e ) {}
            long isbn = Long.parseLong( books[4] );
            double price = Double.parseDouble( books[5] );
            Currency currency = Currency.getInstance( books[6] );
            int copies = Integer.parseInt( books[7] );
            Book book = new Book(books[0], books[1], date, books[3], isbn);
            StockInterface book2 = new Stock(book, copies, price, currency);
            collection.add(book2);
        }
    }
}
```

Esta clase como podemos observar que utiliza extends, donde Catalog extiende Bookcollection para añadir funcionalidad. No tiene ningún atributo es decir está vacio y únicamente inicializamos el constructor **Catalog**. En el constructor, tenemos un for loop donde va leyendo todos los libros que tenemos en el archivo books.xml. Va guardando un stock interface para cada libro, ya que la cardinalidad que tienen stock y libro es 1:1. Luego va añadiendo a collection cada stock interface con los valores. En resumen, crea una hashlist.

CLASE SHOPPINGCART

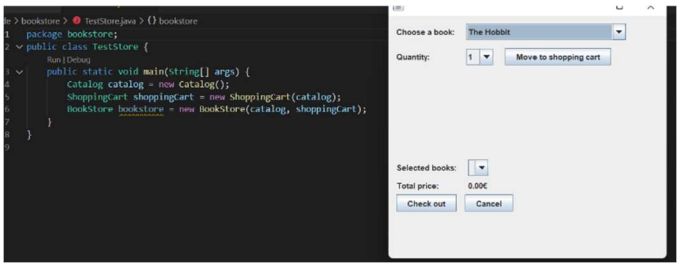
```java
package bookstore;
import java.util.Currency;

public class ShoppingCart extends BookCollection implements ShoppingCartInterface {
    private Catalog catalog;
    public ShoppingCart(Catalog catinit) {
        this.catalog = catinit;//Inicializo el catalogo de la tienda
        //Creo la coleccion que sera mi carrito añadiendo una instancia de stock por cada libro del catalogo.
        for (StockInterface books: catalog.collection){
            StockInterface books_on_cart = new Stock(books.getBook(), 0, books.getPrice(), books.getCurrency());
            collection.add(books_on_cart);
        }
    }
    @Override
    public void addCopies(int numberOfCopies, String booktitle){
        //Primero elimino las copias del catalogo de la tienda
        for(StockInterface books : catalog.collection){
            if(books.getBooktitle().equals(booktitle)){
                books.removeCopies(numberOfCopies);
            }
        }
        //Ahora añado las copias al carrito de la compra
        for(StockInterface books : collection){
            if(books.getBooktitle().equals(booktitle)){
                books.addCopies(numberOfCopies);
            }
        }
    }
```

```java
@Override
public void removeCopies(int numberOfCopies, String booktitle){
    //Primero sumo a la collection de la bookcollection el numero de copias que elimino del carro
    for(StockInterface books : catalog.collection){
        if(books.getBooktitle().equals(booktitle)){
            books.addCopies(numberOfCopies);
        }
    }
    //Elimino copias de mi catalogo que es el carrito
    for(StockInterface books : collection){
        if(books.getBooktitle().equals(booktitle)){
            books.removeCopies(numberOfCopies);
        }
    }
}
public double totalPrice(){
    double price = 0;
    for(StockInterface books_cart :collection){
        price += books_cart.totalPrice();
    }
    return price;
}
public String checkout(){
    Payment payment = new Payment();
    Currency currency = Currency.getInstance("EUR");
    return payment.doPayment(102954432, "cardHolder", totalPrice(), currency);
}
```

Esta clase como podemos observar que utiliza extends, donde ShoppinCart extiende Bookcollection para añadir funcionalidad. Está compuesta por un atributo privado llamado **Catalog.** Además, por el hecho de utilizar extend, sobre escribe bastantes métodos. En primer lugar, tenemos el constructor **ShoppinCart**, donde inicializamos el catálogo de la tienda, y creamos la colección que será nuestro carrito añadiendo una instancia de stock por cada libro del catálogo. También tenemos un método llamado **addCopies,** donde primero eliminamos las copias del catálogo de la tienda y después añadimos las copias al carrito de la compra. Sin embargo, en el método **removeCopies** justo hace lo contrario que el método anterior, suma a la colección de la bookcollection el número de copias que elimina del carro y luego elimina las copias del catálogo que es el carrito. Por otro lado tenemos dos métodos más que son **totalPrice(),** que como dice su nombre, nos devuelve el precio total del shopping cart, y **checkout(),** donde se crea un nuevo pago y devuelve lo que hay que pagar y como hay que pagar.

TEST STORE



Hemos creado para TestStore para poder comprobar que todo el código escrito antes funcione bien y correctamente.

CONCLUSIÓN

Desde el primer momento que empezamos a trabajar en este proyecto supimos que el hecho de comprobar el código sería la clave para crear un código funcional, entendible y limpio. Por el método de prueba error hemos podido modular las funciones correctamente, aunque no ha sido a la primera, ya que hemos tenido que solucionar varios errores en cada clase propuesta.

Nos comimos la cabeza con la clase catalog, ya que, en el enunciado de la práctica aparecía que era una clase void y no supimos entenderlo correctamente. Después de hacerle varias consultas al profesor, varias discusiones entre nosotros e intentarlo una y otra vez de forma repentina, supimos sacarlo adelante. Esta clase ha sido la que nos ha parecido un rompecabezas. Las demás clases hemos tenido errores simples que supimos resolver rápidamente.

Esta práctica ha sido todo un reto para nosotros, debido a que teníamos que compaginar el estudio para los exámenes finales y hacer esta práctica, que nos ha llevado bastante tiempo, pero a pesar de todo supimos sobre ponernos gracias a la buena organización que llevamos a cabo.

Después de todo, podemos decir que hemos producido un código de calidad, producto de las horas de trabajo bien hechas, trabajo en equipo y el método fallo error.