E-Bookstore Management System Report

# Introduction

The Great Ebooks Store needs to acquire a full-service software solution that improves the existing customer experience and the functionality of internal business processes. The fundamental features of the proposed system, called the E-Bookstore Management System, involve ebook management, customer information, shopping cart, order handling, flexible price, discounted regular price, payment processing, and invoices. This paper presents the various stages in creating an E-Bookstore Management System, including design, implementation, testing, and lessons learned while considering the guidelines of object-oriented principles and UML in explaining the system's structure.

# UML Class Diagram and Description

To effectively structure the E-Bookstore Management System, a UML class diagram was created to illustrate the primary components and relationships within the system. The UML diagram consists of classes shown below:

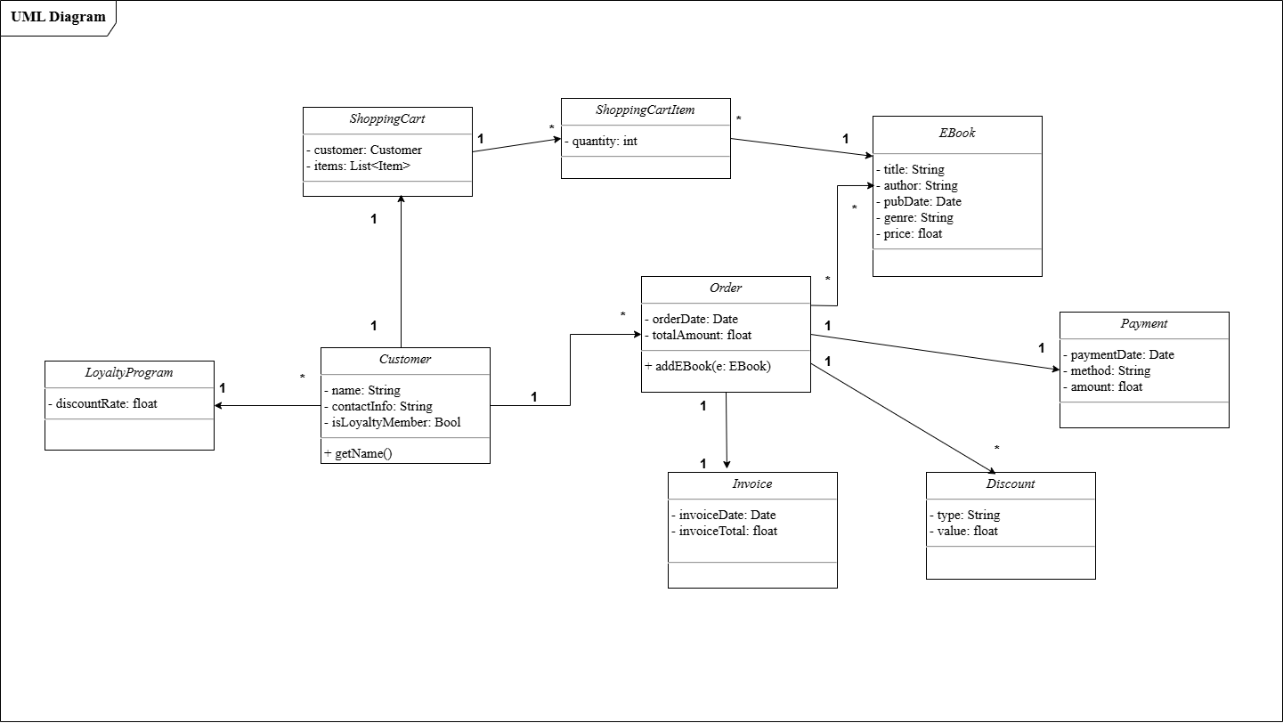


Figure 1:UML Diagram for Ebook management

Every class has properties and behaviors that reflect its position in the system, and how classes are interconnected is as reasonable as possible. Ebook classes work as the basis for the ebook catalog; they contain all the information about each ebook, including the title, author, publication date, genre, and price. The Order has an aggregation relationship with it, meaning that many ebooks can be ordered all at once, but each will have its existence in the catalog. The Customer class defines those users who interact with the system and contains fields such as name, e-mail, phone number, and loyalty level. Every customer may have several orders, but each is bound to have one ShoppingCart; on the other hand, they have many-to-one relation to LoyaltyProgram, as customers are allowed to be members of the program and get personalized benefits.

The Order class identifies the purchase details and has the attributes of order date and total quantity. An order may have several ebooks at once, which is a many-to-many relationship with the EBook class. Only one Order belongs to one Invoice and Payment to store the billing and transaction information. Additionally, an Order could have several associated Discount objects, which would portray any kind of loyalty or bulk purchase discount available, establishing, therefore, a many-to-one relationship between Order and Discount.

The ShoppingCart class has a one-to-one relationship with the Customers. It has a composition relationship with ShoppingCartItem, where each item entails a selected ebook, and the quantity is maintained. Every ShoppingCartItem instance is related to an EBook so that the items in the cart are connected to items available in the catalog. The LoyaltyProgram class controls discount rates for members and has a many-to-one relationship with the Customer class.

# Python Classes Implementation

Based on the recommendation of using Python to implement this system due to its rich Library support and robust Object-Oriented Programming paradigm, the E-Bookstore Management system was developed, translating each class defined in the UML diagram into a modular and well-object-oriented implementation. Classes, namely EBook, Customer, Order, ShoppingCart, ShoppingCartItem, Discount, Invoice, Payment, and LoyaltyProgram, were designed by creating private attributes, getter and setter methods, and \_\_str\_\_ for the main reason of protecting the data and presenting it in the best way possible.

For example, the EBook class includes fields to keep the ebook's title, author, publishing date, genre, and price. Techniques present in this class enable quick access and modification of these attributes while ensuring that data is protected from unauthorized access. Similar precautions are taken in the Customer class, where data about a person, as well as the current status of his/her loyalty card, are to be protected. Basic functionality includes adding new orders and updating the account details.

With the help of the Order class, customers can place orders as they call all the ebooks, total the quantities, and, if necessary, use discounts. Depending on the type of sale and booking order, the Order creates an Invoice containing the item costs, any reduced price or a tax calculation, and a Payment showing the mode of payment, date, and final transaction sum. The ShoppingCart class is a temporary library for ebooks before checkout; it also has functionalities for putting or deleting items into/ from the library and changing quantities. Also, the Discount and LoyaltyProgram classes are intact to ensure that customers derive some benefits from volume and loyalty discounts and thus improve their satisfaction.

## Customer:

class Customer:

    def \_\_init\_\_(self, name, contact\_info, is\_loyalty\_member=False):

        self.\_\_name = name

        self.\_\_contact\_info = contact\_info

        self.\_\_is\_loyalty\_member = is\_loyalty\_member

        self.orders = []  # List to hold orders associated with this customer

    # Getters

    @property

    def name(self):

        """Gets the customer's name."""

        return self.\_\_name

    @property

    def contact\_info(self):

        """Gets the customer's contact information."""

        return self.\_\_contact\_info

    @property

    def is\_loyalty\_member(self):

        """Checks if the customer is a loyalty program member."""

        return self.\_\_is\_loyalty\_member

    # Setters

    @is\_loyalty\_member.setter

    def is\_loyalty\_member(self, status):

        """Sets the customer's loyalty program membership status."""

        self.\_\_is\_loyalty\_member = status

    @contact\_info.setter

    def contact\_info(self, new\_contact\_info):

        """Updates the customer's contact information."""

        self.\_\_contact\_info = new\_contact\_info

    # Methods

    def add\_order(self, order):

        self.orders.append(order)

    # String representation

    def \_\_str\_\_(self):

        return (f"Customer(Name: {self.\_\_name}, Contact Info: {self.\_\_contact\_info}, "

                f"Loyalty Member: {self.\_\_is\_loyalty\_member}, Orders: {len(self.orders)})")

    # Method to display full details as a dictionary (optional)

    def get\_details(self):

        return {

            "Name": self.\_\_name,

            "Contact Info": self.\_\_contact\_info,

            "Loyalty Member": self.\_\_is\_loyalty\_member,

            "Orders Count": len(self.orders)

        }

## Discount:

class Discount:

"""

Class to represent discounts applied on an order.

"""

def \_\_init\_\_(self, discount\_type, discount\_value):

self.\_\_discount\_type = discount\_type

self.\_\_discount\_value = discount\_value

def apply\_discount(self, total\_amount):

return total\_amount \* (1 - self.\_\_discount\_value)

def \_\_str\_\_(self):

return f"Discount(Type: {self.\_\_discount\_type}, Value: {self.\_\_discount\_value \* 100}%)"

## Ebook:

class EBook:

"""

Class to represent an EBook in the store catalog.

Attributes:

title (str): The title of the e-book.

author (str): The author of the e-book.

publication\_date (str): The publication date of the e-book.

genre (str): The genre of the e-book.

price (float): The price of the e-book.

"""

def \_\_init\_\_(self, title, author, publication\_date, genre, price):

"""

Initializes an instance of the EBook class with given attributes.

Args:

title (str): The title of the e-book.

author (str): The author of the e-book.

publication\_date (str): The publication date of the e-book.

genre (str): The genre of the e-book.

price (float): The price of the e-book.

"""

self.\_\_title = title

self.\_\_author = author

self.\_\_publication\_date = publication\_date

self.\_\_genre = genre

self.\_\_price = price

# Getters

@property

def title(self):

"""Gets the title of the e-book."""

return self.\_\_title

@property

def author(self):

"""Gets the author of the e-book."""

return self.\_\_author

@property

def publication\_date(self):

"""Gets the publication date of the e-book."""

return self.\_\_publication\_date

@property

def genre(self):

"""Gets the genre of the e-book."""

return self.\_\_genre

@property

def price(self):

"""Gets the price of the e-book."""

return self.\_\_price

# Setters

@title.setter

def title(self, new\_title):

"""Sets a new title for the e-book."""

self.\_\_title = new\_title

@price.setter

def price(self, new\_price):

"""Sets a new price for the e-book."""

if new\_price < 0:

raise ValueError("Price cannot be negative.")

self.\_\_price = new\_price

# String representation

def \_\_str\_\_(self):

"""

Returns a string representation of the EBook instance.

Returns:

str: A formatted string with e-book details.

"""

return (f"EBook(Title: {self.\_\_title}, Author: {self.\_\_author}, "

f"Publication Date: {self.\_\_publication\_date}, Genre: {self.\_\_genre}, "

f"Price: ${self.\_\_price:.2f})")

# Method to display full details as a dictionary (optional)

def get\_details(self):

"""

Returns all details of the e-book as a dictionary.

Returns:

dict: A dictionary with all e-book details.

"""

return {

"Title": self.\_\_title,

"Author": self.\_\_author,

"Publication Date": self.\_\_publication\_date,

"Genre": self.\_\_genre,

"Price": self.\_\_price

}

## Invoice:

class Invoice:

"""

Class to represent an Invoice generated from an Order.

"""

def \_\_init\_\_(self, invoice\_date, invoice\_total):

self.\_\_invoice\_date = invoice\_date

self.\_\_invoice\_total = invoice\_total

def \_\_str\_\_(self):

return f"Invoice(Date: {self.\_\_invoice\_date}, Total: ${self.\_\_invoice\_total:.2f})"

## Loyalty Program

class LoyaltyProgram:

"""

Class to represent the Loyalty Program details.

"""

def \_\_init\_\_(self, member\_discount=0.10):

self.\_\_member\_discount = member\_discount

def apply\_loyalty\_discount(self, total\_amount):

return total\_amount \* (1 - self.\_\_member\_discount)

def \_\_str\_\_(self):

return f"LoyaltyProgram(Discount: {self.\_\_member\_discount \* 100}%)"

## Order:

from invoice import Invoice

class Order:

"""

Class to represent an Order by a customer.

"""

def \_\_init\_\_(self, customer, order\_date):

self.\_\_order\_date = order\_date

self.\_\_total\_amount = 0

self.\_\_ebooks = []

self.\_\_customer = customer

self.invoice = None

def add\_ebook(self, ebook):

self.\_\_ebooks.append(ebook)

self.\_\_total\_amount += ebook.price

def calculate\_total\_with\_discount(self, discount):

return self.\_\_total\_amount \* (1 - discount)

def generate\_invoice(self):

self.invoice = Invoice(self.\_\_order\_date, self.\_\_total\_amount)

def \_\_str\_\_(self):

ebooks\_list = ", ".join(str(ebook) for ebook in self.\_\_ebooks)

return f"Order(Date: {self.\_\_order\_date}, Total: ${self.\_\_total\_amount:.2f}, EBooks: [{ebooks\_list}])"

## Payment:

class Payment:

"""

Class to represent a Payment for an order.

"""

def \_\_init\_\_(self, payment\_date, payment\_method, amount):

self.\_\_payment\_date = payment\_date

self.\_\_payment\_method = payment\_method

self.\_\_amount = amount

def \_\_str\_\_(self):

return f"Payment(Date: {self.\_\_payment\_date}, Method: {self.\_\_payment\_method}, Amount: ${self.\_\_amount:.2f})"

## Shopping Cart:

from shopping\_cart\_item import ShoppingCartItem

class ShoppingCart:

"""

Class to represent a Shopping Cart.

"""

def \_\_init\_\_(self, customer):

self.\_\_customer = customer

self.\_\_items = []

def add\_item(self, ebook, quantity=1):

item = ShoppingCartItem(ebook, quantity)

self.\_\_items.append(item)

def \_\_str\_\_(self):

items\_list = ", ".join(str(item) for item in self.\_\_items)

return f"ShoppingCart(Customer: {self.\_\_customer.name}, Items: [{items\_list}])"

## Shopping Cart Item:

class ShoppingCartItem:

"""

Class to represent an Item in the Shopping Cart.

"""

def \_\_init\_\_(self, ebook, quantity):

self.\_\_ebook = ebook

self.\_\_quantity = quantity

@property

def ebook(self):

return self.\_\_ebook

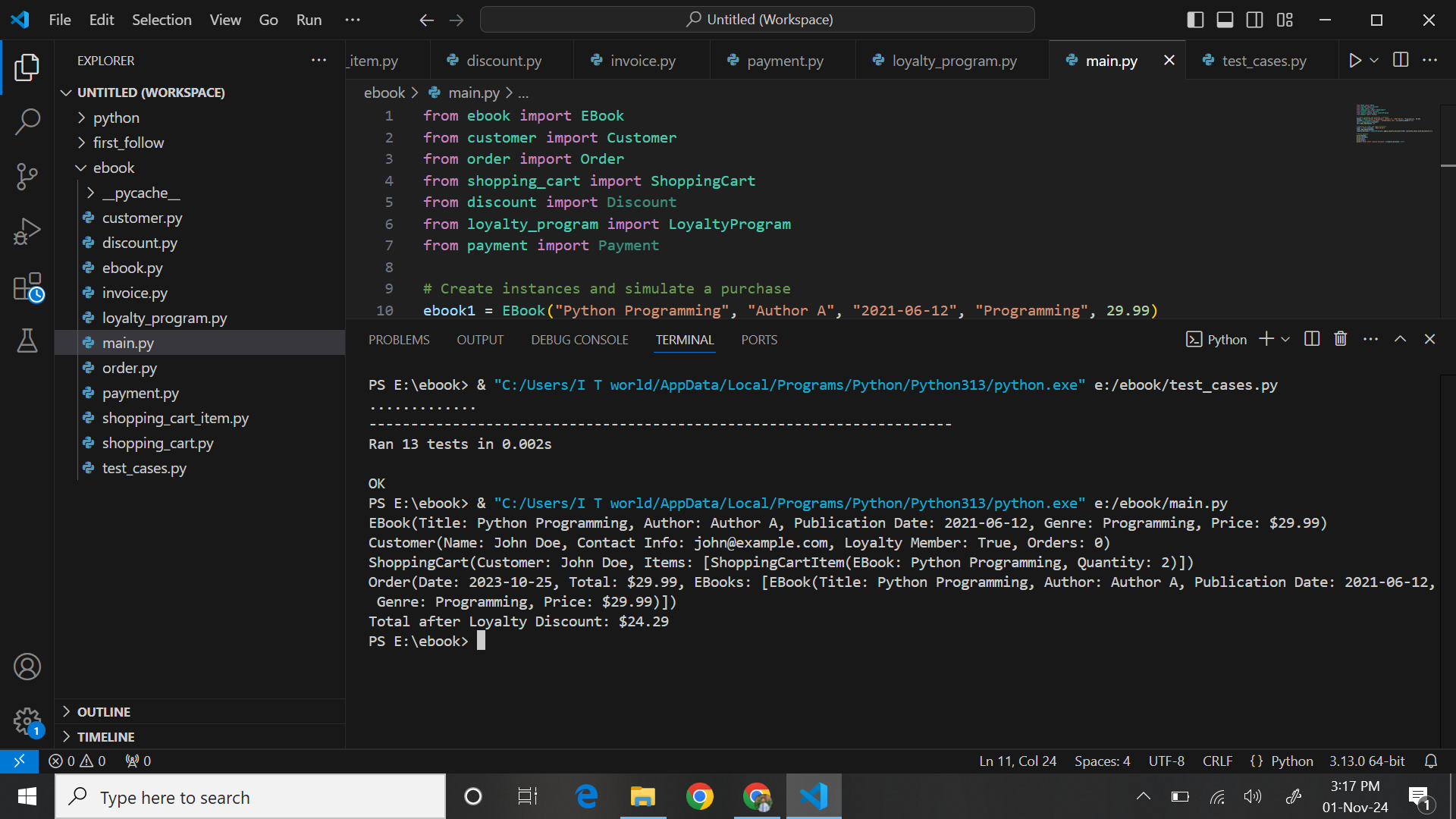
@property

def quantity(self):

return self.\_\_quantity

def \_\_str\_\_(self):

return f"ShoppingCartItem(EBook: {self.\_\_ebook.title}, Quantity: {self.\_\_quantity})"



# Test Cases

Since the goal of the E-Bookstore Management System was to be fully functional and reliable, additional test cases were created for the actual application using Python’s unit test. Every VST can involve checking one or several aspects of functionality within the system to increase the accuracy and reliability of each component. The various test cases under each functionality are detailed below

for better understanding.

## Ebook Management

The examples below were developed to ensure the basic functionalities of the class EBook were tested. Some of the tests involved adding new ebooks to the catalog, changing the details of existing ebooks, and deleting ebooks from the catalog. These tests prove that getters and setters should be used to keep the data in the catalog consistent and point out that other tests help to check that the catalog is kept sufficiently recent.

## Customer Account Management

The class was created for customers to ensure the proper creation of accounts for customers and to check on updated information about clients and their loyalty program membership. These tests ensure customer data is correctly stored in the system and the representation of whether a customer is eligible for a loyalty program and, therefore, should be allowed a member’s discount.

## Shopping Cart Operations

The ShoppingCart class testing also aims for customers to be able to add some items to the cart, change the quantity of these items if needed, and, in general, delete the items from the cart. Every test case ensures that a cart functions correctly: the totality is correct, and items represent selected ebooks correctly. Furthermore, ShoppingCartItem tests also verify the relationship with the EBook instances, meaning the items currently in the cart are the right kind.

## Discount Application

These simple price tests must also be conducted to ensure that they are applied correctly to orders with loyalty and bulk purchase discounts. These tests have proved that loyalty program members are offered a 10% discount on all purchases, while wholesale orders of ebooks of eight and above are given a 20% discount. These test cases are vital for getting the proper price and determining the customer's feelings.

## Order Processing and Invoice Generation

Sample examinations for the Order class help to verify that the orders are correctly stored and match the correct ebooks, order date, and total amount. To ensure that details of the prices per item, additional discounts, and the totals are also well displayed to the customers, the Invoice generation upon order completion was also subjected to this test. These tests are crucial for transparency and customer satisfaction because customers obtain insights about the products received.

## Payment Processing

Payment tests were performed to check the Payment instance handling payment details, including the payment type, date, and amount. These tests confirm the effectiveness of the payment processing system in recording all transactions appropriately and securely.

## Test Case define on python:

import unittest

from ebook import EBook

from customer import Customer

from order import Order

from shopping\_cart import ShoppingCart

from shopping\_cart\_item import ShoppingCartItem

from discount import Discount

from invoice import Invoice

from loyalty\_program import LoyaltyProgram

from payment import Payment

class TestEBookStore(unittest.TestCase):

def setUp(self):

# Initialize common test objects used across multiple test cases

self.ebook1 = EBook("Python Programming", "Author A", "2021-06-12", "Programming", 29.99)

self.ebook2 = EBook("Data Science Fundamentals", "Author B", "2020-01-20", "Data Science", 39.99)

self.customer = Customer("John Doe", "john@example.com", is\_loyalty\_member=True)

self.cart = ShoppingCart(self.customer)

self.order = Order(self.customer, "2023-10-25")

def test\_add\_ebook\_to\_catalog(self):

"""Test adding an e-book to the catalog"""

self.assertEqual(self.ebook1.title, "Python Programming")

self.assertEqual(self.ebook1.price, 29.99)

def test\_modify\_ebook\_details(self):

"""Test modifying details of an e-book"""

self.ebook1.price = 24.99 # modifying price with setter

self.assertEqual(self.ebook1.price, 24.99)

def test\_remove\_ebook(self):

"""Test removing an e-book from the catalog (simulation by deleting object)"""

ebook = EBook("Sample Book", "Author C", "2021-01-01", "Sample Genre", 19.99)

del ebook # deleting the ebook object

with self.assertRaises(NameError):

print(ebook) # accessing after deletion should raise NameError

def test\_add\_customer\_account(self):

"""Test adding a new customer account"""

new\_customer = Customer("Alice Smith", "alice@example.com")

self.assertEqual(new\_customer.name, "Alice Smith")

self.assertFalse(new\_customer.is\_loyalty\_member)

def test\_modify\_customer\_account(self):

"""Test modifying customer account details"""

self.customer.is\_loyalty\_member = False

self.assertFalse(self.customer.is\_loyalty\_member)

def test\_add\_ebook\_to\_shopping\_cart(self):

"""Test adding e-books to the shopping cart"""

self.cart.add\_item(self.ebook1, 2)

self.assertEqual(len(self.cart.\_ShoppingCart\_\_items), 1) # One item added

self.assertEqual(self.cart.\_ShoppingCart\_\_items[0].quantity, 2)

def test\_remove\_item\_from\_cart(self):

"""Test removing items from the shopping cart"""

self.cart.add\_item(self.ebook1, 1)

self.cart.\_ShoppingCart\_\_items.pop(0) # remove the item

self.assertEqual(len(self.cart.\_ShoppingCart\_\_items), 0)

def test\_apply\_loyalty\_discount(self):

"""Test applying loyalty discount to an order"""

self.order.add\_ebook(self.ebook1)

loyalty\_discounted\_total = LoyaltyProgram().apply\_loyalty\_discount(self.order.calculate\_total\_with\_discount(0))

expected\_total = self.ebook1.price \* 0.9 # 10% loyalty discount applied

self.assertAlmostEqual(loyalty\_discounted\_total, expected\_total)

def test\_bulk\_purchase\_discount(self):

"""Test applying a bulk discount for 8 or more e-books in a single order"""

for \_ in range(8):

self.order.add\_ebook(self.ebook1)

total\_with\_bulk\_discount = self.order.calculate\_total\_with\_discount(0.2)

expected\_total = 8 \* self.ebook1.price \* 0.8 # 20% off bulk purchase

self.assertAlmostEqual(total\_with\_bulk\_discount, expected\_total)

def test\_invoice\_generation(self):

"""Test the generation of an invoice with discounts applied"""

self.order.add\_ebook(self.ebook1)

self.order.add\_ebook(self.ebook2)

self.order.generate\_invoice()

self.assertIsInstance(self.order.invoice, Invoice)

self.assertAlmostEqual(self.order.invoice.\_Invoice\_\_invoice\_total, 69.98)

def test\_payment\_processing(self):

"""Test payment processing for an order"""

payment = Payment("2023-10-25", "Credit Card", 69.98)

self.assertEqual(payment.\_Payment\_\_amount, 69.98)

self.assertEqual(payment.\_Payment\_\_payment\_method, "Credit Card")

def test\_discount\_application(self):

"""Test the application of discounts"""

discount = Discount("Loyalty Discount", 0.1)

discounted\_total = discount.apply\_discount(100.0)

self.assertAlmostEqual(discounted\_total, 90.0) # 10% off

def test\_cart\_total\_amount(self):

"""Test calculation of the total amount in the cart before checkout"""

self.cart.add\_item(self.ebook1, 2)

total\_amount = sum(item.ebook.price \* item.quantity for item in self.cart.\_ShoppingCart\_\_items)

self.assertAlmostEqual(total\_amount, 59.98) # 2 copies of ebook1 at 29.99 each

if \_\_name\_\_ == "\_\_main\_\_":

unittest.main()

# Github repository link

# Summary of Learnings

This paper notes that creating the E-Bookstore Management System helped me understand the principles underlying object-oriented software design and development. The project helped me learn a lot about the best ways of modeling real-life objects and their relationships with the help of classical UML class diagrams. Determining one-to-many and many-to-many relationships let me detail the reforming system architecture in a manner that accurately mirrors business circumstances. It also promoted the creation of reusable components while maintaining effective scalability.

The use of object-oriented programming principles in the context of Python acted as a triumphant reminder of encapsulation, inheritance, and modularity. To ensure high modularity and code security, I separated each class into different files and used private attributes with getter and setter methods. This approach does more than ease the process of maintaining the code and making the codebase comprehensible; it also offers a solution for boosting the reliability of the data and minimizing the possibility of making a mistake within the context of a system.

I found how to write comprehensive test cases to be the most impressive lesson since it underlined that for code to be believed to work correctly, its functionality must be verified. I understood the importance of making assertions, which help identify the expected results, which will allow finding out one's problem early enough and ensure that all parts run as suspected under varying circumstances. Testing also underlined the importance of coverage since, during the development process, all features, including ebooks management or payment accepting functionality, were validated. This helped improve the system's stability to a great extent

; the time spent writing down the code and creating this official report has helped the author understand the value of precise language in information technology. It helps share and discuss with others because whenever someone wants to know something about a project, they refer to what has been documented. The choice transformed my writing skills and augmented the production of the technical details into a comprehensible report for different categories of the audience.

# Conclusion

The E-Bookstore Management System successfully fulfills the specifications of the Great Ebooks Store, with its effectiveness, flexibility, and aesthetics to handle ebooks, customer info, carts, coupons, and purchases. As the system model was designed, implemented, and tested, the system has shown high reliability and functionality. Outsourcing through the planning tool UML, the development with Python has offered a reliable model that has given a suitable software solution. This project also provides the intended objectives but brings awareness about the importance of systematic engineering methodology, starting from the designing, testing, and documentation phases.