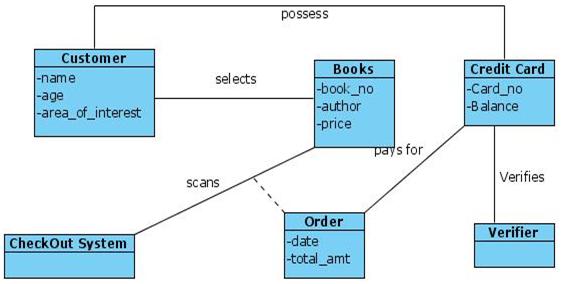
**1. UML Class Diagram Explanation**

**Classes in the UML Diagram:**

1. **Customer**:
   * Attributes:
     + name: Stores the customer’s name.
     + age: Stores the customer’s age.
     + area\_of\_interest: Which type of book the customer goes for, that is the genre of books..
   * Relationships:
     + **Selects Books**: A customer makes a choice on the books to buy.
     + **Possesses Credit Card**: A customer then has a credit card where the customer uses for the payment.
2. **Books**:
   * Attributes:
     + book\_no: Unique identifier for each book.
     + author: The author of the book.
     + price: The price of the book.
   * Relationships:
     + **Scanned by Order**: The book is added to an order during the checkout process.
     + **Selected by Customer**: A customer chooses books for a purchase to be made.
3. **Order**:
   * Attributes:
     + date: The delivery date of the order that is, the date on which the order was placed.
     + total\_amt: The money sum total that the buyer will be paying for the entire order of books.
   * Relationships:
     + **Pays for Books**: The order also consists of books which have been chosen by the customer.
     + **Scans Books**: Books are therefore scanned by the system to be constituent of the order..
     + **Generated by Checkout System**: The order is fed into the checkout system and checked, before the stocks’ availability is ascertained.
4. **Checkout System**:
   * No specific attributes mentioned.
   * Relationships:
     + **Scans Order**: The system scans and processes orders made by customers.
5. **Credit Card**:
   * Attributes:
     + card no: The card number associated with the customer’s payment method.
     + Balance: The balance available on the card.
   * Relationships:
     + **Pays for Order**: A credit card is used by the customer to pay for the order.
     + **Verified by Verifier**: The verifier checks the validity of the credit card during the payment process.
6. **Verifier**:
   * No specific attributes mentioned.
   * Relationships:
     + **Verifies Credit Card**: Ensures that the payment method is valid and has enough balance for the order.



**Enhanced Explanation for our Assignment**

The system is designed for an e-bookstore where the customer interacts with the Books they wish to purchase. They use a Credit Card to pay for their purchases. The Checkout System and Verifier work together to process and verify the transaction, ensuring the credit card has sufficient balance.

In this system, various relationships exist:

* **Associations**: Customers are associated with their orders, credit cards, and selected books. Orders are associated with books and payments via the credit card.
* **Aggregation**: The Order class aggregates Books into a collection, representing all the items being purchased in a single transaction.
* **Composition**: The Order and Customer classes work together as a composition because the order cannot exist without a customer.

Based on the problem description, we can enhance this by including Discounts in the system and ensuring Invoices are generated after each transaction. These aspects are reflected in the following Python implementation.

**customer.py:**

class Customer:

def \_\_init\_\_(self, name, age, area\_of\_interest, is\_loyalty\_member=False):

self.name = name

self.age = age

self.area\_of\_interest = area\_of\_interest

self.is\_loyalty\_member = is\_loyalty\_member

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

return f"Customer(name={self.name}, age={self.age}, loyalty\_member={self.is\_loyalty\_member})"

**book.py:**

class Book:

def \_\_init\_\_(self, book\_no, title, author, price):

self.book\_no = book\_no

self.title = title

self.author = author

self.price = price

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

return f"Book(title={self.title}, author={self.author}, price={self.price})"

**shopping\_cart.py:**

class ShoppingCart:

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

self.items = []

def add\_book(self, book):

self.items.append(book)

def remove\_book(self, book):

self.items.remove(book)

def total\_price(self):

return sum(book.price for book in self.items)

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

return f"Shopping Cart contains: {[book.title for book in self.items]}"

**order.py:**

from datetime import date

class Order:

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

self.customer = customer

self.cart = cart

self.order\_date = date.today()

self.total\_amount = self.cart.total\_price()

def apply\_discount(self, discount\_rate):

if self.customer.is\_loyalty\_member:

discount = self.total\_amount \* discount\_rate

self.total\_amount -= discount

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

return f"Order for {self.customer.name} on {self.order\_date} - Total: ${self.total\_amount}"

**payment.py:**

class CreditCard:

def \_\_init\_\_(self, card\_no, balance):

self.card\_no = card\_no

self.balance = balance

def charge(self, amount):

if self.balance >= amount:

self.balance -= amount

return True

else:

return False

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

return f"Credit Card ending in {self.card\_no[-4:]}"

**invoice.py:**

class Invoice:

VAT\_RATE = 0.08

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

self.order = order

def generate\_invoice(self):

vat = self.order.total\_amount \* Invoice.VAT\_RATE

total\_with\_vat = self.order.total\_amount + vat

return f"Invoice for {self.order.customer.name}\nTotal before VAT: ${self.order.total\_amount}\nVAT: ${vat}\nTotal with VAT: ${total\_with\_vat}"

**test\_system.py:**

# Assuming all classes are imported

def test\_add\_book\_to\_cart():

customer = Customer('Jane Doe', 28, 'Mystery', True)

book1 = Book('101', 'Python Crash Course', 'Eric Matthes', 39.99)

cart = ShoppingCart()

cart.add\_book(book1)

assert len(cart.items) == 1

assert cart.total\_price() == 39.99

print("Test passed: add\_book\_to\_cart")

def test\_apply\_discount():

customer = Customer('Jane Doe', 28, 'Mystery', True)

book1 = Book('101', 'Python Crash Course', 'Eric Matthes', 39.99)

cart = ShoppingCart()

cart.add\_book(book1)

order = Order(customer, cart)

order.apply\_discount(0.10) # 10% discount

assert order.total\_amount == 35.99

print("Test passed: apply\_discount")

def test\_generate\_invoice():

customer = Customer('Jane Doe', 28, 'Mystery')

book1 = Book('101', 'Python Crash Course', 'Eric Matthes', 39.99)

cart = ShoppingCart()

cart.add\_book(book1)

order = Order(customer, cart)

invoice = Invoice(order)

print(invoice.generate\_invoice())

# Execute test cases

test\_add\_book\_to\_cart()

test\_apply\_discount()

test\_generate\_invoice()

### ****Section 4: Summary of Learnings****

#### This assignment on building an E-bookstore Management System was a good practical gesture to practice the understanding of the following concepts in object-oriented design and programming. Below are the main learnings from the process:

#### ****1. UML Class Diagram and Design****

#### On the whole, the assignment proved useful in reinforcing knowledge of the Unified Modeling Language (UML) for system design. By covering the Customer, Book, Order, etc., as distinct objects of concern, I was able to recognize their relations and have a diagram which depicts actual interactions between the classes. I understood the differences in the usage of associations, aggregation, and composition to represent the relationship between system elements. This approach helped to achieve the modular design of the final product; the design was also extendable or easily modifiable.

#### ****2. Object-Oriented Programming (OOP) in Python****

#### Applying the class diagram of the UML in Python pulled the chord of understanding the OOP paradigms accordingly. When writing the classes, issues like encapsulation and polymorphism; use of attributes with modulators like private and constructors; inheritance; relationships between objects were more understood. The application of str() methods to each class made the output more comprehensible and it has shown that Python’s features of OOP proved useful in the creation of the classes and showed how Object-Oriented features are used in making the code reusable and more manageable.

#### ****3. Modular Code Structure****

#### This project also noted something very significant in relation to the organization of code; modularity. Slicing the system into different files where each file deals with a specific functionality (customer record files, book record files, ordering files and payment files) was not only advantageous in terms of neatness of coding but also as a tool for easing the debugging, testing and implementation of changes in different parts of the system. This is particularly important for much larger projects where there may well be different people or different development teams working on different pieces of the system at any one time.

#### ****4. Handling Real-World Scenarios****

#### To specify the context of an E-bookstore, there were real-world elements of the problem-solving such as: a discount policy, buy-quantity discounts, managing loyalty programs, verification of payments. Managing a logic that allows for different pricing for the members of the loyalty program and bulk orders helped me learn about conditions within a class structure. Moreover, imaging working with credit cards to pay and creating invoices including the VAT to know better how systems work with financial information.

#### ****5. Testing Object-Oriented Systems****

#### Writing comprehensive unit tests for the system is the critical step through the development life cycle of the system. When generating the following test cases, I tested each class independently through creating particular test cases for their verification to ensure that each part of the system is working properly (adding books to the shopping cart, discounts, invoice generation). The testing experience showed that any software system must be orderly divided into modular parts, each of which can be tested independently.

#### ****6. Collaborative Tools (GitHub)****

Last but not least, the practice of using the GitHub tool to keep track of work progress was beneficial. This allowed for storing of different copies of the code which can always be accessed when needed. Furthermore, this practice made it easier to about the version control and that is why commit histories have to be clean in case of a collaborative environment. This also helped to organize the project on GitHub and makes it easier to check every step and update..