1. **Classes, Attributes, and Methods**

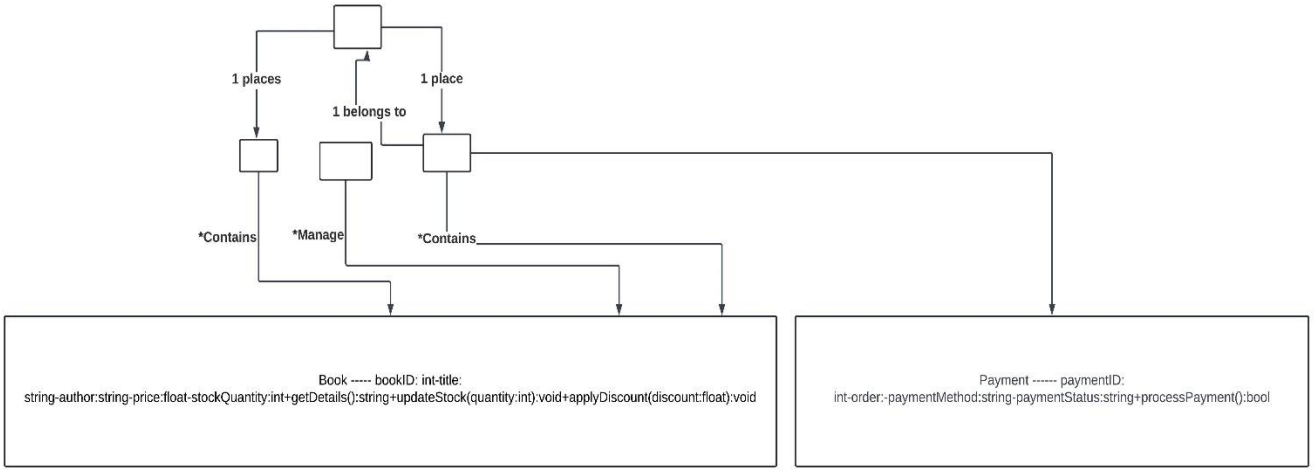
Class	Attributes	Methods
Customer	customerID: int, name: string, email: string, password: string, address: string, phoneNumber: string, orderHistory: List<Order>	register(): void, login(email: string, password: string): bool, logout(): void, searchBook(keyword: string): List<Book>, addToCart(Book, quantity: int): void, placeOrder(): Order, trackOrder(orderID: int): Order, updateProfile(newDetails: Customer): void
Book	bookID: int, title: string, author: string, genre: string, ISBN: string, price: float, stockQuantity: int, description: string, publisher: string	getDetails(): string, checkAvailability(): bool, updateStock(newStock: int): void, applyDiscount(discount: float): void
Cart	cartID: int, customer: Customer, cartItems: List<CartItem>	addItem(Book, quantity: int): void, removeItem(Book): void, calculateTotal(): float, clearCart(): void, checkout(): Order
CartItem	cartItemID: int, book: Book, quantity: int, price: float	getItemTotal(): float, updateQuantity(newQuantity: int): void
Order	orderID: int, orderDate: date, customer: Customer, orderItems: List<OrderItem>, shippingMethod: string, totalAmount: float, orderStatus: string	calculateTotal(): float, updateStatus(newStatus: string): void, trackOrder(): string, cancelOrder(): void
OrderItem	orderItemID: int, book: Book, quantity: int, price: float	calculateItemTotal(): float
Payment	paymentID: int, paymentMethod: string, paymentStatus: string, amount: float	processPayment(): bool, refundPayment(): bool
Shipping	shippingID: int, shippingMethod: string, shippingCost: float, shippingAddress: string, shippingStatus: string	calculateShippingCost(): float, updateShippingStatus(newStatus: string): void
StockManager	managerID: int, name: string	checkStock(book: Book): int, updateStock(book: Book, newQuantity: int): void, reorderBook(book: Book, quantity: int): void
Review	reviewID: int, customer: Customer, book: Book, rating: int, comment: string	addReview(customer: Customer, book: Book, rating: int, comment: string): void, editReview(newRating: int, newComment: string): void

2. **Class Diagram**

The class diagram would show relationships such as:

- Customer has a Cart and can place multiple Orders.
- Order contains multiple OrderItems, each representing a Book.
- Order has a Payment and Shipping associated with it.
- Book is referenced in CartItem, OrderItem, and Review.
- StockManager manages the Book stock.

3. UML Class Diagram



Part 2: Discuss the benefits and limitations of using class diagrams as a modeling tool for developing the above system. Please include access modifiers.

Benefits of Using Class Diagrams for Modeling the Book Ordering System

- Clear Structure and Organization

Class diagrams offer a structured, visual representation of system components, relationships, and interactions. In this book ordering system, the class diagram helps to clearly define entities like Customer, Book, Cart, Order, Payment, and StockManager, making it easier for developers to understand and navigate.

- Encapsulation and Access Control

Class diagrams can represent access modifiers (e.g., + for public, - for private), enforcing encapsulation. For instance, private attributes like - customerID or - email in the Customer class

ensure that sensitive data is accessible only through designated public methods. This highlights security measures early in the design process.

- Identifies Responsibilities and Methods

Defining class-specific methods clarifies each class's responsibilities. For example, Customer methods like searchBook() and addToCart() establish the functionalities expected from each class, helping developers adhere to principles like the Single Responsibility Principle (SRP).

- Supports Scalability and Future Expansion

Class diagrams provide a flexible framework that can accommodate changes and additions, such as expanding with new classes for other system modules (e.g., adding Discount or Shipping classes). This scalability reduces the complexity of modifications during development.

- Easy Collaboration and Communication

For teams, class diagrams act as a common visual language that simplifies communication among developers, project managers, and stakeholders. They make system design more accessible, allowing non-technical team members to follow system requirements and discuss potential changes.

- Facilitates Code Generation

Class diagrams can assist in auto-generating code structure. Many IDEs allow importing UML diagrams to create skeleton code, reducing the need for manual implementation and helping developers maintain consistency with the initial design.

Limitations of Using Class Diagrams for Modeling the System

- Limited Dynamic Interaction Representation

Class diagrams focus on static structure and do not effectively capture dynamic interactions or runtime behavior between objects. This limitation may hinder understanding complex interactions in real-time, such as order tracking and cart updates. Sequence diagrams or activity diagrams might be needed to represent such dynamic behaviors.

- Potential Over-Simplification

Class diagrams simplify relationships and interactions, which may lead to an oversimplified view of complex system dependencies. For example, this model does not account for potential failures in Payment processing, which might need additional handling not captured in a static class representation.

- Not Ideal for Detailed Access Control

While access modifiers (e.g., + for public, - for private) indicate basic encapsulation, class diagrams do not fully capture nuanced security details such as role-based access or permission levels, which may be required in a more complex ordering system involving administrators and customers with different access rights.

- Difficulty in Handling Non-Class-Based Relationships

Class diagrams are best suited for object-oriented systems and may struggle to represent non-object-oriented relationships or external dependencies, such as those with databases, external APIs, or web services.

- May Not Reflect Performance Considerations

Class diagrams focus on structural aspects and rarely account for performance or scalability implications, such as load balancing, caching strategies, or data handling for a high number of concurrent users. These considerations might need separate documentation or architectural diagrams to be accurately represented.

- Scalability Challenges for Large Systems

As the system grows in complexity, class diagrams may become overcrowded, making them difficult to read and manage. Detailed diagrams with many classes or complex relationships can reduce readability, defeating the purpose of visual simplicity in UML diagrams.