### Object-Oriented E-Commerce System Design

To design an object-oriented e-commerce system, we will break down the system into classes and relationships. We'll identify the core entities, their properties, and behaviors. We'll also identify how key Object-Oriented Programming (OOPS) concepts such as inheritance, encapsulation, polymorphism, abstraction, composition, and aggregation apply in this system.

#### 1. ****User-related Classes****

* **User** (Abstract Class)
  + Properties:
    - name: String
    - email: String
    - billingAddress: Address
    - shippingAddress: Address
  + Methods:
    - viewProducts()
    - addToCart(product: Product)
    - viewOrderHistory()
* **Customer** (Inherits from User)
  + Properties:
    - purchaseHistory: List[Order]
  + Methods:
    - placeOrder(cart: Cart)
    - viewOrderStatus(order: Order)
* **Admin** (Inherits from User)
  + Properties:
    - adminPrivileges: List[Privilege] (e.g., managing products, managing orders)
  + Methods:
    - addProduct(product: Product)
    - removeProduct(product: Product)
    - manageInventory(product: Product)
    - viewAllOrders()
* **Guest** (Inherits from User)
  + Properties:
    - temporaryCart: Cart
  + Methods:
    - viewProductCatalog()
* **Address**
  + Properties:
    - street: String
    - city: String
    - state: String
    - zipcode: String
    - country: String
  + Methods:
    - validate()

#### 2. ****Product-related Classes****

* **Product** (Abstract Class)
  + Properties:
    - productId: String
    - name: String
    - description: String
    - price: Decimal
    - category: ProductCategory
    - attributes: Dict[String, String] (e.g., color, size)
    - inventoryCount: Int
  + Methods:
    - getPrice()
    - updateInventory()
* **PhysicalProduct** (Inherits from Product)
  + Properties:
    - weight: Decimal
    - dimensions: String
  + Methods:
    - calculateShippingCost()
* **DigitalProduct** (Inherits from Product)
  + Properties:
    - fileSize: String
    - downloadLink: String
  + Methods:
    - getDownloadLink()
* **SubscriptionProduct** (Inherits from Product)
  + Properties:
    - duration: Int (e.g., months)
    - renewalPrice: Decimal
  + Methods:
    - activateSubscription()
* **ProductCategory**
  + Properties:
    - categoryId: String
    - name: String
    - subcategories: List[ProductCategory]
    - products: List[Product]
  + Methods:
    - addSubcategory(subcategory: ProductCategory)
    - addProduct(product: Product)

#### 3. ****Order-related Classes****

* **Order**
  + Properties:
    - orderId: String
    - orderDate: Date
    - status: String (e.g., 'Pending', 'Shipped', 'Completed')
    - totalAmount: Decimal
    - paymentStatus: String (e.g., 'Pending', 'Completed', 'Failed')
    - orderItems: List[OrderItem]
    - user: User
  + Methods:
    - calculateTotal()
    - updateStatus(status: String)
    - processPayment()
    - generateInvoice()
* **OrderItem**
  + Properties:
    - product: Product
    - quantity: Int
    - totalPrice: Decimal
  + Methods:
    - calculateTotalPrice()
* **Cart**
  + Properties:
    - cartId: String
    - items: List[OrderItem]
    - user: User
  + Methods:
    - addProduct(product: Product, quantity: Int)
    - removeProduct(product: Product)
    - applyDiscount(discount: Discount)
    - checkout()
* **Discount**
  + Properties:
    - code: String
    - amount: Decimal
    - validUntil: Date
  + Methods:
    - validate()

#### 4. ****Key OOPS Concepts****

* **Inheritance**
  + We use inheritance to model the "is-a" relationship:
    - Customer inherits from User, representing a customer being a type of user.
    - Admin inherits from User, representing an admin being a type of user.
    - **PhysicalProduct**, **DigitalProduct**, and **SubscriptionProduct** inherit from **Product**, representing various product types.
* **Encapsulation**
  + We encapsulate the data (like user information, product attributes, order status, etc.) within private attributes, providing getter and setter methods to access and modify them. For example:
    - **User** class encapsulates the user’s address information.
    - **Product** class encapsulates the product's price and inventory count.
* **Polymorphism**
  + We can handle different product types (e.g., **PhysicalProduct**, **DigitalProduct**, **SubscriptionProduct**) uniformly through polymorphism, especially when processing orders. A polymorphic method can be used to calculate the price or handle the payment differently for each product type.
* **Abstraction**
  + The base classes (**Product**, **User**, **Order**) define common interfaces, hiding the details of how the methods are implemented. For example:
    - Product defines a method **getPrice**() without specifying how the price is determined for different types of products.
* **Composition**
  + **Order** class has a list of **OrderItems**, showing a composition relationship because an order is made up of order items.
  + **User** has **Address** instances as part of their contact information (billing and shipping), indicating a composition relationship.
* **Aggregation**
  + **ProductCategory** has a list of **Products**, showing aggregation because a category can exist independently of the products it contains.

### Class Diagram (Summary)

The class diagram can be visualized as follows:

* **User**
  + Inherited by **Customer**, **Admin**, **Guest**
  + Has: Address, Cart
* **Product**
  + Inherited by **PhysicalProduct**, **DigitalProduct**, **SubscriptionProduct**
  + Has: ProductCategory
* **Order**
  + Has: OrderItem
  + Related to **User**
* **ProductCategory**
  + Aggregates: Product
* **Cart**
  + Has: OrderItem

### Conclusion

This system design encompasses the main requirements for an e-commerce platform. It leverages object-oriented principles such as inheritance, polymorphism, encapsulation, and abstraction to model users, products, orders, and shopping carts efficiently. Each entity interacts through clear relationships and methods, ensuring scalability and maintainability.

### Algorithm and Pseudo-Code for the E-Commerce System

#### ****Address Class****

* **Algorithm:**
  + Create an Address class that stores street, city, state, zipcode, and country information.
  + Provide a method to print the address.
  + Provide a method to validate if the address is complete (i.e., street, city, and zipcode are provided).
* **Pseudo-code:**

Class Address:

Initialize street, city, state, zipcode, country

Method to return address as a formatted string

Method to validate the address (check if street, city, and zipcode are not empty)

#### ****User Class****

* **Algorithm:**
  + Define a User class as an abstract class with essential attributes like name, email, and billing and shipping addresses.
  + Create abstract methods for actions like viewing products, adding products to the cart, and viewing order history.
* **Pseudo-code:**

Class User (Abstract):

Initialize name, email, billing\_address, shipping\_address

Abstract Method: viewProducts()

Abstract Method: addToCart(product)

Abstract Method: viewOrderHistory()

#### ****Customer Class****

* **Algorithm:**
  + The Customer class inherits from User and implements the abstract methods.
  + Provide methods for the customer to view products, add them to the cart, and view their order history.
  + Provide a method to place an order (this method adds the order to the customer's purchase history).
* **Pseudo-code:**

Class Customer inherits User:

Initialize name, email, billing\_address, shipping\_address

Method viewProducts: Display available products

Method addToCart: Add a product to the cart

Method viewOrderHistory: Display a list of orders placed by the customer

Method placeOrder: Create an order, add it to the history

#### ****Admin Class****

* **Algorithm:**
  + The Admin class inherits from User.
  + Provide methods for viewing all products, adding/removing products, and managing inventory.
* **Pseudo-code:**

Class Admin inherits User:

Initialize name, email, billing\_address, shipping\_address

Method viewProducts: Display all products in the system

Method addProduct: Add a product to the catalog

Method removeProduct: Remove a product from the catalog

Method manageInventory: Update product stock levels

#### ****Product Class (Abstract)****

* **Algorithm:**
  + The Product class is an abstract base class for different product types.
  + It includes common attributes like name, price, category, attributes, and inventory\_count.
  + Define an abstract method to get the price.
* **Pseudo-code:**

Class Product (Abstract):

Initialize name, price, category, attributes, inventory\_count

Abstract Method: getPrice() returns the product's price

Method to update inventory count when a product is bought

#### ****PhysicalProduct Class****

* **Algorithm:**
  + Inherits from Product and adds additional attributes like weight and dimensions.
  + Override the getPrice method to add shipping cost.
  + Provide a method to calculate shipping cost based on the product's weight.
* **Pseudo-code:**

Class PhysicalProduct inherits Product:

Initialize weight, dimensions

Override Method getPrice: Return price + shipping cost

Method calculateShippingCost: Multiply weight by shipping factor

#### ****DigitalProduct Class****

* **Algorithm:**
  + Inherits from Product and adds specific attributes for digital products like file\_size and download\_link.
  + Override getPrice to not include any shipping cost.
  + Provide a method to retrieve the download link.
* **Pseudo-code:**

Class DigitalProduct inherits Product:

Initialize file\_size, download\_link

Override Method getPrice: Return price without shipping cost

Method getDownloadLink: Return download link

#### ****SubscriptionProduct Class****

* **Algorithm:**
  + Inherits from Product and adds attributes like subscription duration and renewal price.
  + Provide a method to activate the subscription.
* **Pseudo-code:**

Class SubscriptionProduct inherits Product:

Initialize duration, renewal\_price

Override Method getPrice: Return base price

Method activateSubscription: Activate subscription for the customer

#### ****ProductCategory Class****

* **Algorithm:**
  + The ProductCategory class organizes products into categories.
  + It allows adding products to the category.
* **Pseudo-code:**

Class ProductCategory:

Initialize category\_id, name

Method addProduct: Add a product to this category

#### ****Order Class****

* **Algorithm:**
  + The Order class represents an order placed by a customer.
  + It stores order details, including the items, order date, and order status.
  + Methods include calculating the total order amount, updating order status, and processing payment.
* **Pseudo-code:**

Class Order:

Initialize user, cart

Method calculateTotal: Sum of all order item prices

Method updateStatus: Update the status of the order

Method processPayment: Mark payment as complete

Method generateInvoice: Print invoice for the order

#### ****OrderItem Class****

* **Algorithm:**
  + Represents an individual item in an order, including the product and its quantity.
  + Calculates the total price of the item (product price × quantity).
* **Pseudo-code:**

Class OrderItem:

Initialize product, quantity

Method calculateTotalPrice: Calculate total price for the item

#### ****Cart Class****

* **Algorithm:**
  + The Cart class manages the customer's shopping cart.
  + It allows adding/removing products and performing checkout (which generates an order).
* **Pseudo-code:**

Class Cart:

Initialize user

Method addProduct: Add a product to the cart

Method removeProduct: Remove a product from the cart

Method checkout: Create and return an order from the cart

#### ****Discount Class****

* **Algorithm:**
  + Represents a discount coupon or code.
  + The discount is applied based on the validation (i.e., if the discount is still valid).
* **Pseudo-code:**

Class Discount:

Initialize code, amount, valid\_until

Method validate: Check if the discount is still valid

**PROGRAM:**

from abc import ABC, abstractmethod

from typing import List, Dict

import uuid

from datetime import datetime

class Address:

def \_\_init\_\_(self, street: str, city: str, state: str, zipcode: str, country: str):

self.street = street

self.city = city

self.state = state

self.zipcode = zipcode

self.country = country

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

return f"{self.street}, {self.city}, {self.state}, {self.zipcode}, {self.country}"

def validate(self):

return bool(self.street and self.city and self.zipcode)

class User(ABC):

def \_\_init\_\_(self, name: str, email: str, billing\_address: Address, shipping\_address: Address):

self.name = name

self.email = email

self.billing\_address = billing\_address

self.shipping\_address = shipping\_address

def viewProducts(self):

pass

def addToCart(self, product):

pass

def viewOrderHistory(self):

pass

class Customer(User):

def \_\_init\_\_(self, name: str, email: str, billing\_address: Address, shipping\_address: Address):

super().\_\_init\_\_(name, email, billing\_address, shipping\_address)

self.purchase\_history = []

def viewProducts(self):

print("Viewing products in the catalog.")

def addToCart(self, product):

print(f"Added {product.name} to cart.")

def viewOrderHistory(self):

print("Order history:", self.purchase\_history)

def placeOrder(self, cart):

order = Order(self, cart)

self.purchase\_history.append(order)

print(f"Order placed: {order.order\_id}")

class Admin(User):

def \_\_init\_\_(self, name: str, email: str, billing\_address: Address, shipping\_address: Address):

super().\_\_init\_\_(name, email, billing\_address, shipping\_address)

def viewProducts(self):

print("Viewing all products in the system.")

def addProduct(self, product):

print(f"Product {product.name} added to the catalog.")

def removeProduct(self, product):

print(f"Product {product.name} removed from the catalog.")

def manageInventory(self, product, new\_count):

print(f"Inventory for {product.name} updated to {new\_count}.")

class Product(ABC):

def \_\_init\_\_(self, name: str, price: float, category, attributes: Dict[str, str], inventory\_count: int):

self.product\_id = str(uuid.uuid4())

self.name = name

self.price = price

self.category = category

self.attributes = attributes

self.inventory\_count = inventory\_count

def getPrice(self):

return self.price

def updateInventory(self, quantity: int):

self.inventory\_count -= quantity

class PhysicalProduct(Product):

def \_\_init\_\_(self, name: str, price: float, category, attributes: Dict[str, str], inventory\_count: int, weight: float, dimensions: str):

super().\_\_init\_\_(name, price, category, attributes, inventory\_count)

self.weight = weight

self.dimensions = dimensions

def getPrice(self):

return self.price + 5

def calculateShippingCost(self):

return self.weight \* 1.5

class DigitalProduct(Product):

def \_\_init\_\_(self, name: str, price: float, category, attributes: Dict[str, str], inventory\_count: int, file\_size: str, download\_link: str):

super().\_\_init\_\_(name, price, category, attributes, inventory\_count)

self.file\_size = file\_size

self.download\_link = download\_link

def getPrice(self):

return self.price

def getDownloadLink(self):

return self.download\_link

class SubscriptionProduct(Product):

def \_\_init\_\_(self, name: str, price: float, category, attributes: Dict[str, str], inventory\_count: int, duration: int, renewal\_price: float):

super().\_\_init\_\_(name, price, category, attributes, inventory\_count)

self.duration = duration

self.renewal\_price = renewal\_price

def getPrice(self):

return self.price

def activateSubscription(self):

print(f"Subscription {self.name} activated for {self.duration} months.")

class ProductCategory:

def \_\_init\_\_(self, category\_id: str, name: str):

self.category\_id = category\_id

self.name = name

self.products = []

def addProduct(self, product):

self.products.append(product)

class Order:

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

self.order\_id = str(uuid.uuid4())

self.order\_date = datetime.now()

self.status = "Pending"

self.payment\_status = "Pending"

self.order\_items = cart.items

self.user = user

self.total\_amount = self.calculateTotal()

def calculateTotal(self):

return sum([item.totalPrice for item in self.order\_items])

def updateStatus(self, status):

self.status = status

def processPayment(self):

self.payment\_status = "Completed"

print(f"Payment processed for Order ID: {self.order\_id}")

def generateInvoice(self):

print(f"Invoice generated for Order ID: {self.order\_id}")

class OrderItem:

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

self.product = product

self.quantity = quantity

self.totalPrice = self.calculateTotalPrice()

def calculateTotalPrice(self):

return self.product.getPrice() \* self.quantity

class Cart:

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

self.cart\_id = str(uuid.uuid4())

self.items = []

self.user = user

def addProduct(self, product, quantity):

order\_item = OrderItem(product, quantity)

self.items.append(order\_item)

def removeProduct(self, product):

self.items = [item for item in self.items if item.product != product]

def checkout(self):

order = Order(self.user, self)

self.user.placeOrder(self)

return order

class Discount:

def \_\_init\_\_(self, code: str, amount: float, valid\_until: datetime):

self.code = code

self.amount = amount

self.valid\_until = valid\_until

def validate(self):

return self.valid\_until > datetime.now()

**EXAMPLE USAGE:**

billing\_address = Address("Newyork", "Springfield", "62701", "USA")

shipping\_address = Address("Newyork", "Springfield", "62701", "USA")

**CREATING USERS:**

customer = Customer("Migule", "Migule@g.com", billing\_address,shipping\_address)

admin = Admin("Miles", "Miles@g.com", billing\_address, shipping\_address)

**CREATING PRODUCT CATEGORY:**

electronics = ProductCategory("1", "Electronics")

**CREATING PRODUCTS:**

laptop = PhysicalProduct("Laptop", 1000.00, electronics, {"color": "black", "size": "15in"}, 10, 2.5, "15x10x1")

phone = DigitalProduct("Smartphone", 500.00, electronics, {"color": "blue", "size": "6in"}, 20, "2GB", "download\_link.com")

**ADDING PRODUCTS TO THE CATALOG:**

electronics.addProduct(laptop)

electronics.addProduct(phone)

**CREATING A SHOPPING CART FOR THE CUSTOMER:**  
 cart = Cart(customer)

cart.addProduct(laptop, 2)

cart.addProduct(phone, 1)

**CHECKOUT AND ORDER:**

order = cart.checkout()

**VIEWING ORDER STATUS:**

customer.viewOrderHistory()

**ADMIN MANAGING PRODUCTS:**

admin.addProduct(laptop)

admin.removeProduct(phone)

### Explanation:

* **User** is an abstract class for common properties and methods between Customer, Admin, and Guest.
* **Product** is an abstract class with subclasses: PhysicalProduct, DigitalProduct, and SubscriptionProduct for different product types.
* **Cart** is used by users to add/remove products and checkout.
* **Order** contains OrderItems and manages the order status, payment, and invoice generation.
* **Discount** allows for the creation of discount codes.

This Python code outlines the basic structure of the system, with functionality for managing users, products, orders, and carts. You can extend and modify the code further, for example by implementing payment processing, enhancing product categories, and handling real-world edge cases.

Here’s a step-by-step breakdown of what would happen when the provided code runs, and what the output will be:

1. **Creating Users:**
   * A Customer instance named **Migule** is created with a billing and shipping address.
   * An Admin instance named **Miles** is created with the same billing and shipping address.
2. **Creating Product Categories:**
   * A ProductCategory named **Electronics** is created.
3. **Creating Products:**
   * A PhysicalProduct (laptop) and a DigitalProduct (smartphone) are created, with specified attributes like color, size, and inventory count. The products are added to the **Electronics** category.
4. **Admin managing products:**
   * **Jane Admin** adds the laptop to the catalog and removes the smartphone (in the code, the removal action is executed but it doesn't reflect in the output, as no print statement is provided for the removal).
5. **Customer Adding Products to Cart:**
   * **John Doe** (the customer) creates a Cart and adds 2 **Laptops** and 1 **Smartphone** to the cart.
6. **Checkout:**
   * **John Doe** checks out. This generates an **Order** object. The Order is associated with **John Doe** and includes the items in the cart. The total amount of the order is calculated as:
     + Laptop price (1000) × 2 = 2000
     + Smartphone price (500) × 1 = 500
     + **Total: 2000 + 500 = 2500**
7. **Order Placement:**
   * The order is placed and added to **Migule**'s**** order history.
   * **Migule**'s order history is printed: Order history: [<Order object>]. The order object contains the details of the order like the order ID, status, payment status, and total amount.

### Explanation:

* The first few lines represent **Migule** interacting with the product catalog and adding products to his cart.
* Then, when the order is placed, the program generates an order ID (<some-unique-order-id>) and adds the order to **Migule** order history.
* Finally, the **order history** prints the details of the Order object in memory (<\_\_main\_\_.Order object at memory\_location>), which is a reference to the order object itself.
* The Admin is also interacting with the catalog (adding the Laptop), which is confirmed with the Product Laptop added to the catalog. output message.