Logic code:

# Import datetime for date and time operations  
from datetime import datetime  
  
# Parent class for personal information  
class Person:  
 def \_\_init\_\_(self, name, contact\_info):  
 self.\_name = name # Store name  
 self.\_contact\_info = contact\_info # Store contact info  
  
 # Getter for name  
 @property  
 def name(self):  
 return self.\_name  
  
 # Setter for name  
 @name.setter  
 def name(self, name):  
 self.\_name = name  
  
 # Getter for contact info  
 @property  
 def contact\_info(self):  
 return self.\_contact\_info  
  
 # Setter for contact info  
 @contact\_info.setter  
 def contact\_info(self, contact\_info):  
 self.\_contact\_info = contact\_info  
  
 # String representation  
 def \_\_str\_\_(self):  
 return f"Person(name={self.\_name}, contact\_info={self.\_contact\_info})"  
  
  
# Customer class inheriting from Person  
class Customer(Person):  
 def \_\_init\_\_(self, name, contact\_info, loyalty\_member=False):  
 super().\_\_init\_\_(name, contact\_info) # Initialize from Person  
 self.\_loyalty\_member = loyalty\_member # Loyalty member status  
 self.\_address = "" # Customer address  
  
 # Getter for loyalty member  
 @property  
 def loyalty\_member(self):  
 return self.\_loyalty\_member  
  
 # Setter for loyalty member  
 @loyalty\_member.setter  
 def loyalty\_member(self, loyalty\_member):  
 self.\_loyalty\_member = loyalty\_member  
  
 # Getter for address  
 @property  
 def address(self):  
 return self.\_address  
  
 # Setter for address  
 @address.setter  
 def address(self, address):  
 self.\_address = address  
  
 # String representation  
 def \_\_str\_\_(self):  
 return f"Customer(name={self.\_name}, loyalty\_member={self.\_loyalty\_member}, address={self.\_address})"  
  
  
# Class for eBook details  
class EBook:  
 def \_\_init\_\_(self, title, author, publication\_date, genre, price):  
 self.\_title = title # Title of the eBook  
 self.\_author = author # Author of the eBook  
 self.\_publication\_date = publication\_date # Publication date  
 self.\_genre = genre # Genre of the eBook  
 self.\_price = price # Price of the eBook  
 self.\_ISBN = "000-0-00-000000-0" # ISBN number  
  
 # Getter for title  
 @property  
 def title(self):  
 return self.\_title  
  
 # Setter for title  
 @title.setter  
 def title(self, title):  
 self.\_title = title  
  
 # Getter for price  
 @property  
 def price(self):  
 return self.\_price  
  
 # Setter for price  
 @price.setter  
 def price(self, price):  
 self.\_price = price  
  
 # Getter for ISBN  
 @property  
 def ISBN(self):  
 return self.\_ISBN  
  
 # Setter for ISBN  
 @ISBN.setter  
 def ISBN(self, ISBN):  
 self.\_ISBN = ISBN  
  
 # String representation  
 def \_\_str\_\_(self):  
 return f"EBook(title={self.\_title}, author={self.\_author}, price={self.\_price}, ISBN={self.\_ISBN})"  
  
  
# Class for shopping cart management  
class ShoppingCart:  
 def \_\_init\_\_(self):  
 self.\_items = [] # List of items in cart  
 self.\_creation\_date = datetime.now() # Creation date of cart  
 self.\_customer\_id = None # ID of customer  
  
 # Add item to cart  
 def add\_item(self, ebook, quantity=1):  
 self.\_items.append((ebook, quantity)) # Add eBook and quantity  
  
 # Remove item from cart  
 def remove\_item(self, ebook):  
 self.\_items = [(item, qty) for item, qty in self.\_items if item != ebook] # Remove specified item  
  
 # Calculate total price  
 def calculate\_total(self):  
 return sum(item.price \* qty for item, qty in self.\_items) # Sum of item prices  
  
 # String representation  
 def \_\_str\_\_(self):  
 items\_str = ', '.join([f"{item.title} (x{qty})" for item, qty in self.\_items]) # List items  
 return f"ShoppingCart(items=[{items\_str}])"  
  
 # Getter for creation date  
 @property  
 def creation\_date(self):  
 return self.\_creation\_date  
  
 # Getter for customer ID  
 @property  
 def customer\_id(self):  
 return self.\_customer\_id  
  
 # Setter for customer ID  
 @customer\_id.setter  
 def customer\_id(self, customer\_id):  
 self.\_customer\_id = customer\_id  
  
  
# Class for order management  
class Order:  
 def \_\_init\_\_(self, customer, items, order\_date):  
 self.\_customer = customer # Customer for order  
 self.\_items = items # Items in order  
 self.\_order\_date = order\_date # Date of order  
 self.\_discount = 0.0 # Initial discount  
 self.apply\_discounts() # Apply discounts  
 self.\_total\_amount = self.calculate\_total() # Total order amount  
 self.\_order\_id = f"ORD-{int(datetime.now().timestamp())}" # Unique order ID  
  
 # Apply discounts based on conditions  
 def apply\_discounts(self):  
 if self.\_customer.loyalty\_member: # Check loyalty member  
 self.\_discount = 0.10 # 10% discount  
 if sum(qty for \_, qty in self.\_items) >= 5: # Bulk purchase discount  
 self.\_discount = max(self.\_discount, 0.20) # Max discount 20%  
  
 # Calculate total after discounts  
 def calculate\_total(self):  
 total = sum(item.price \* qty for item, qty in self.\_items) # Sum item prices  
 total \*= (1 - self.\_discount) # Apply discount  
 self.\_total\_amount = total # Set total amount  
 return total  
  
 # String representation  
 def \_\_str\_\_(self):  
 return f"Order(customer={self.\_customer.name}, total\_amount={self.calculate\_total()})"  
  
 # Getter for order ID  
 @property  
 def order\_id(self):  
 return self.\_order\_id  
  
  
# Class for invoice generation  
class Invoice:  
 VAT\_RATE = 0.08 # Fixed VAT rate  
  
 def \_\_init\_\_(self, order):  
 self.\_order = order # Order associated with invoice  
 self.\_VAT = order.calculate\_total() \* Invoice.VAT\_RATE # Calculate VAT  
 self.\_final\_amount = order.calculate\_total() + self.\_VAT # Final total with VAT  
 self.\_invoice\_number = f"INV-{int(datetime.now().timestamp())}" # Unique invoice number  
  
 # Generate invoice as a string  
 def generate\_invoice(self):  
 return f"Invoice for {self.\_order.\_customer.name}: Total = {self.\_final\_amount}, VAT = {self.\_VAT}, Invoice Number = {self.\_invoice\_number}"  
  
 # String representation  
 def \_\_str\_\_(self):  
 return self.generate\_invoice()  
  
 # Getter for invoice number  
 @property  
 def invoice\_number(self):  
 return self.\_invoice\_number  
  
 # Getter for VAT  
 @property  
 def VAT(self):  
 return self.\_VAT  
  
 # Getter for final amount  
 @property  
 def final\_amount(self):  
 return self.\_final\_amount

Test code:

import unittest # Import unittest for testing framework  
from A2 import EBook, Customer, ShoppingCart, Order, Invoice # Import classes from A2 module  
from datetime import datetime # Import datetime for date and time  
  
  
# Test class for EBook System  
class TestEBookSystem(unittest.TestCase):  
  
 # Test adding an eBook  
 def test\_add\_ebook(self):  
 ebook = EBook("Fahrenheit 451", "Ray Bradbury", "2007-11-01", "Fiction", 29.99) # Create eBook  
 print(f"Created EBook: {ebook}")  
  
 self.assertEqual(ebook.title, "Fahrenheit 451") # Assert title is correct  
 self.assertEqual(ebook.price, 29.99) # Assert price is correct  
 print("EBook title and price are correct.")  
  
 # Test creating a customer  
 def test\_customer\_creation(self):  
 customer = Customer("Raya", "Raya@gmail.com", True) # Create Customer  
 print(f"Created Customer: {customer}")  
  
 self.assertEqual(customer.name, "Raya") # Assert name is correct  
 self.assertTrue(customer.loyalty\_member) # Assert loyalty status is correct  
 print("Customer name and loyalty status are correct.")  
  
 # Test calculating shopping cart total  
 def test\_shopping\_cart\_total(self):  
 ebook1 = EBook("Book 1", "Author 1", "2007-01-01", "Sci-Fi", 15.99) # Create first eBook  
 ebook2 = EBook("Book 2", "Author 2", "2007-02-01", "Fantasy", 25.99) # Create second eBook  
 cart = ShoppingCart() # Create shopping cart  
 cart.add\_item(ebook1, 2) # Add two copies of ebook1  
 cart.add\_item(ebook2, 1) # Add one copy of ebook2  
 print(f"Shopping Cart Items: {[f'{item.title} (x{qty})' for item, qty in cart.\_items]}")  
  
 calculated\_total = cart.calculate\_total() # Calculate total price in cart  
 print(f"Calculated Shopping Cart Total: {calculated\_total}")  
 self.assertAlmostEqual(calculated\_total, 57.97, places=2) # Assert total is correct  
 print("Shopping Cart total is correct.")  
  
 # Test discount application in order  
 def test\_order\_discount(self):  
 customer = Customer("Raya", "Raya@gmail.com", True) # Create loyalty customer  
 ebook = EBook("Fahrenheit 451", "Ray Bradbury", "2007-11-01", "Non-Fiction", 20.0) # Create eBook  
 items = [(ebook, 5)] # List with eBook and quantity 5  
 order = Order(customer, items, datetime.now()) # Create order with items and customer  
 order.apply\_discounts() # Apply discounts to order  
 print(f"Order after applying discounts: {order}")  
  
 calculated\_total = order.calculate\_total() # Calculate total with discounts  
 print(f"Calculated Order Total with Discounts: {calculated\_total}")  
 self.assertAlmostEqual(calculated\_total, 80.0, places=2) # Assert total with discount  
 print("Order total with discounts is correct.")  
  
 # Test invoice generation  
 def test\_invoice\_generation(self):  
 customer = Customer("Raya", "Raya@gmail.com") # Create customer  
 ebook = EBook("Fahrenheit 451", "Ray Bradbury", "2007-11-01", "Non-Fiction", 20.0) # Create eBook  
 items = [(ebook, 1)] # List with one eBook  
 order = Order(customer, items, datetime.now()) # Create order with customer and items  
 invoice = Invoice(order) # Create invoice for order  
 generated\_invoice = invoice.generate\_invoice() # Generate invoice string  
 print(f"Generated Invoice: {generated\_invoice}")  
  
 self.assertIn("Invoice for Raya", generated\_invoice) # Assert invoice contains correct text  
 print("Invoice generation is correct.")  
  
  
# Run tests if script is executed directly  
if \_\_name\_\_ == "\_\_main\_\_":  
 unittest.main()