Hamda Sami Bukattara

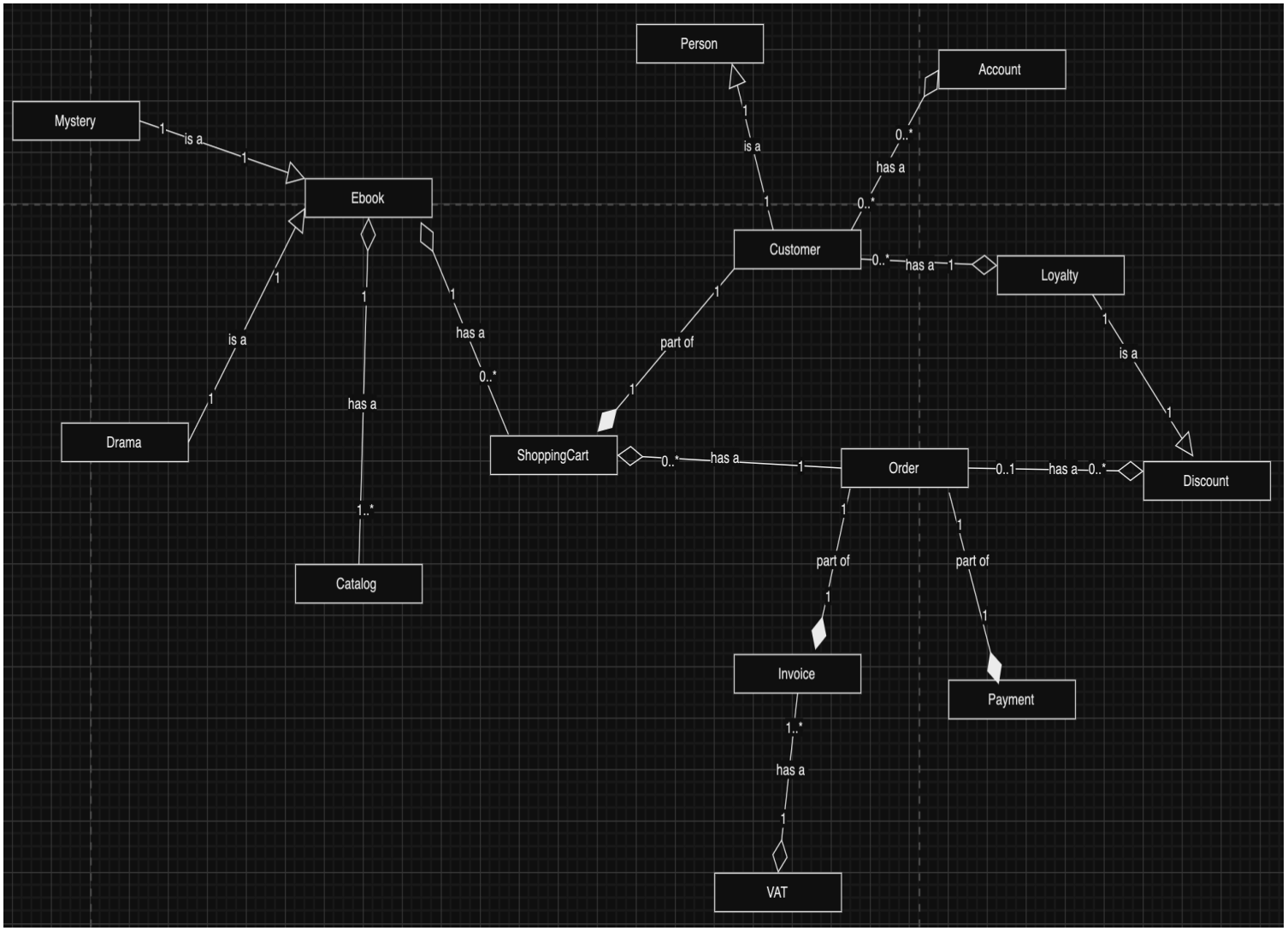
College of Interdisciplinary Studies, Zayed University

ICS220: Program. Fund.

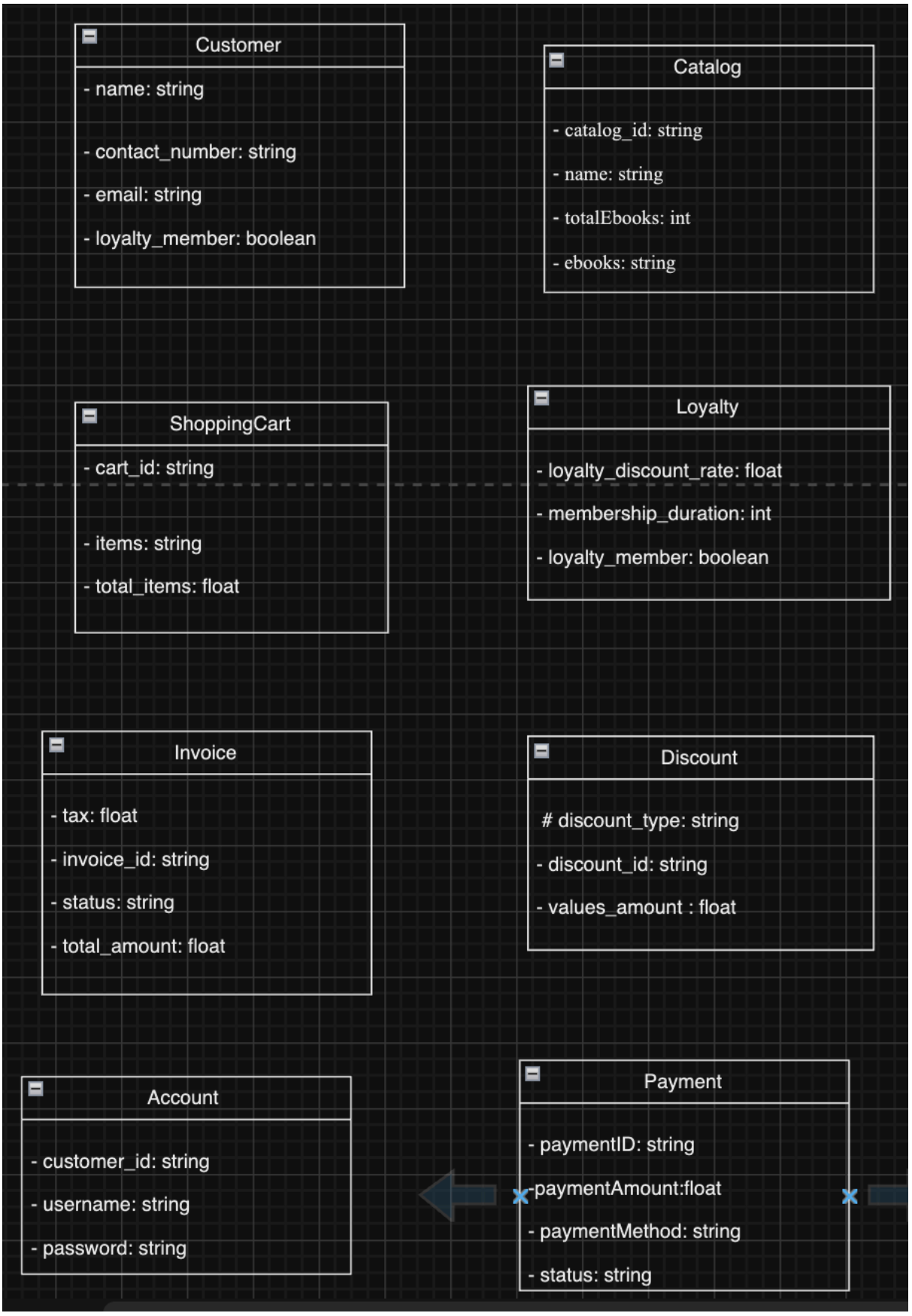
Dr. Sujith Mathew

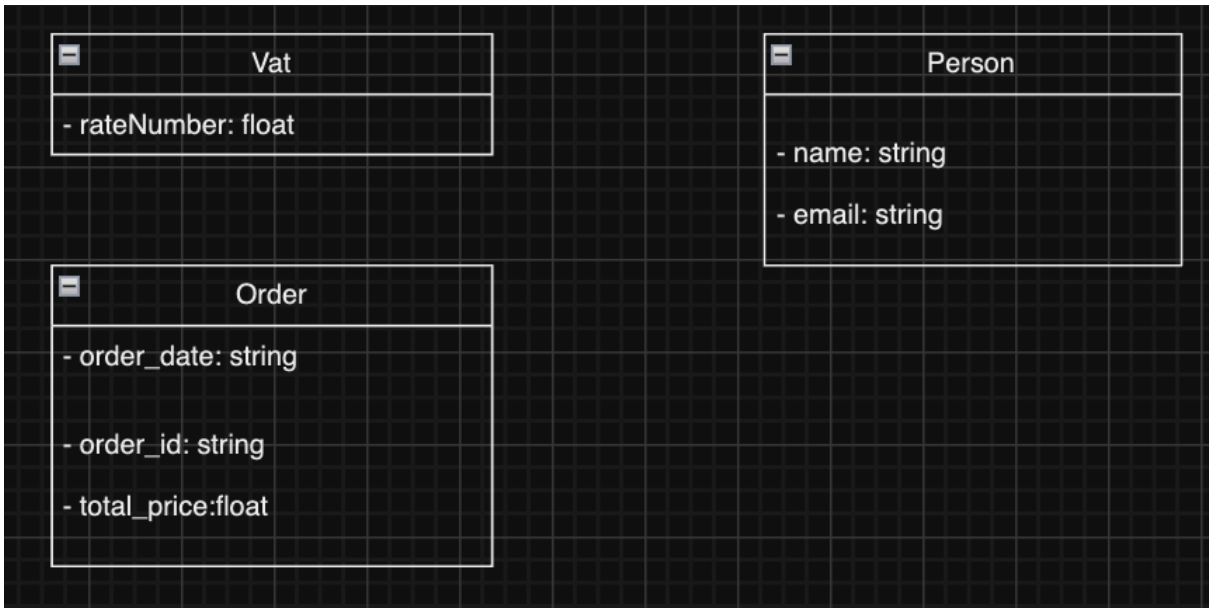
November 4, 2024

1) Design UML Class Diagram:



2) Classes and their attributes:


Customer
- name: string
- contact_number: string
- email: string
- loyalty_member: boolean
Catalog
- catalog_id: string
- name: string
- totalEbooks: int
- ebooks: string
ShoppingCart
- cart_id: string
- items: string
- total_items: float
Loyalty
- loyalty_discount_rate: float
- membership_duration: int
- loyalty_member: boolean
Invoice
- tax: float
- invoice_id: string
- status: string
- total_amount: float
Discount
discount_type: string
- discount_id: string
- values_amount : float
Account
- customer_id: string
- username: string
- password: string
Payment
- paymentID: string
-paymentAmount:float
- paymentMethod: string
- status: string



3) Explanation why I picked the relations:

Inheritance Arrows:

Inheritance relationships use the "is a" phrase, indicating that one class is a subtype of another and inherits its properties.

- **Mystery and Ebook:**

Description: Mystery is a type of Ebook. This means Mystery inherits the characteristics of Ebook but has additional attributes specific to mystery books.

Explanation: By defining Mystery as an Ebook, it can share common Ebook properties like author and title and add unique details for mystery books.

Cardinality: 1 on both sides, as each Mystery instance is a single Ebook.

Example: A new Mystery book, "The Mysterious Island," is a specific type of Ebook.

- **Drama and Ebook:**

Description: Drama is a type of Ebook. It inherits from Ebook but includes characteristics unique to the drama genre.

Explanation: Drama books are treated as Ebooks, but they can have extra attributes specific to drama, allowing them to be categorized separately if needed.

Cardinality: 1 on both sides, as each Drama instance is a single Ebook.

Example: "Romeo and Juliet" is a Drama and also an Ebook, so it’s listed under both categories.

- **Customer and Person:**

Description: Customer is a type of Person, meaning every Customer has all the properties of a Person but with additional customer specific information.

Explanation: Customers can use general Person details like name and email and add extra customer-related features, making it easier to manage customer data.

Cardinality: 1 on both sides, as each Customer is a Person.

Example: Adding "John Doe" as a Customer automatically includes him as a Person with additional customer details.

- **Loyalty and Discount:**

Description: Loyalty is a type of Discount, meaning Loyalty programs inherit from Discount and add loyalty specific features.

Explanation: This structure allows Loyalty programs to use general discount rules while adding exclusive discounts for loyalty members.

Cardinality: 1 on both sides, as each Loyalty is a Discount.

Example: A "10% Loyalty Discount" is a specialized Discount only for loyal customers.

Composition Arrows:

Composition relationships are described with "has a," indicating that one class contains another in a dependent, essential relationship.

- **Customer and ShoppingCart:**

Description: Each Customer has a ShoppingCart, which holds items they plan to buy.

Explanation: Since each Customer has their own ShoppingCart, if the Customer is removed, their ShoppingCart is also deleted. The ShoppingCart cannot exist without its Customer.

Cardinality: 1 near Customer, 1 near ShoppingCart, as each Customer has a unique ShoppingCart.

Example: If Customer "Sara" creates a ShoppingCart and deletes her account later, her ShoppingCart is also deleted.

- **Order and Invoice:**

Description: Each Order has a unique Invoice detailing the purchase, and the Invoice cannot exist without the Order.

Explanation: Each Order has a specific Invoice, and if the Order is canceled, the Invoice is also removed.

Cardinality: 1 near Order, 1 near Invoice, as each Order has a unique Invoice.

Example: An order for "The Great Gatsby" generates an Invoice specifically for that Order; if the order is canceled, the Invoice is also deleted.

- **Order and Payment:**

Description: Each Order has a Payment that completes the purchase.

Explanation: The Payment is directly linked to the Order, if the Order is removed, the Payment is also removed.

Cardinality: 1 near Order, 1 near Payment, as each Order has a unique Payment.

Example: "Order #172" has a Payment made by credit card; if the Order is refunded, the Payment record is also removed.

Aggregation Arrows:

Aggregation relationships are described with "part of," indicating a weaker association where parts can exist independently of the whole.

- **Catalog and Ebook:**

Description: A Catalog has a collection of Ebooks, but Ebooks can exist independently and are not dependent on the Catalog.

Explanation: The Catalog acts as a container for Ebooks, allowing them to be organized but not requiring them to exist only in the Catalog.

Cardinality: 1..* near Catalog, 1 near Ebook, meaning a Catalog can contain multiple Ebooks.

Example: "Pride and Prejudice" is part of a Catalog, but if the Catalog is deleted, the Ebook still exists.

- **ShoppingCart and Order:**

Description: A ShoppingCart may contain multiple Orders that are part of the cart. Orders can still exist if removed from the ShoppingCart.

Explanation: The ShoppingCart loosely organizes Orders, but removing the ShoppingCart does not delete existing Orders.

Cardinality: 0..* near ShoppingCart, 1 near Order, as a ShoppingCart can contain multiple Orders.

Example: A ShoppingCart might include past Orders, but if the cart is cleared, those Orders remain unaffected.

- **Customer and Account:**

Description: A Customer has a relationship with one or more Accounts, but Accounts are not dependent on the Customer for existence.

Explanation: Accounts can be linked to a Customer without needing to be deleted if the Customer is removed.

Cardinality: 0..* near Customer, 1 near Account, as a Customer can have multiple Accounts.

Example: A Customer has a billing account and a loyalty account that are part of the customer's profile but can exist independently.

- **Customer and Loyalty:**

Description: A Customer can participate in a Loyalty program, but the Loyalty program is not dependent on the Customer.

Explanation: Customers can join or leave Loyalty programs, and the program continues to exist independently.

Cardinality: 0..* near Customer, 1 near Loyalty, as multiple Customers can join the same Loyalty program.

Example: A Customer joins a "10% off" Loyalty program that is part of their benefits, but if they leave, the program remains available to others.

- **Order and Discount:**

Description: An Order can apply a Discount, which is part of the Order to reduce its total cost. Discounts are optional and can exist independently.

Explanation: Discounts apply as needed, allowing reuse across multiple Orders without being tied to any specific Order.

Cardinality: 0..1 near Order, 0..* near Discount, meaning an Order can have zero or one Discount.

Example: An Order may use a "Holiday Discount" that is part of its price reduction, but if no discount is available, the Order proceeds normally.

- **Invoice and VAT:**

Description: An Invoice has a VAT entry to calculate tax. VAT details are part of the invoice but can exist independently.

Explanation: Each Invoice includes VAT to calculate tax, but VAT details are reusable across multiple Invoices.

Cardinality: 1..* near Invoice, 1 near VAT, as each Invoice has one or more VAT entries.

Example: An Invoice includes VAT as part of the total, but VAT can also apply to other invoices independently.

- **Ebook and ShoppingCart:**

Description: A ShoppingCart has a collection of Ebooks, which are part of the cart. Ebooks can exist independently if removed from the cart.

Explanation: Ebooks are temporarily part of the ShoppingCart, and removing them doesn't affect their availability in the system.

Cardinality: 0..* near ShoppingCart, 1 near Ebook, meaning a ShoppingCart can contain multiple Ebooks.

Example: If a customer adds "The Hobbit" to the cart, it is part of the cart temporarily but remains available even if removed.

4) Class code :

```
# Class 1 - Ebook
# I created an Ebook class to store details about ebooks.
class Ebook:
    def __init__(self, author, title, genre, price, publication_date):
        # I used these attributes to hold the basic information of an ebook.
        self.author = author  # The author of the ebook
        self.title = title  # The title of the ebook
        self.genre = genre  # The genre of the ebook
        self.price = price  # The price of the ebook
        self.publication_date = publication_date  # The publication date of
the ebook

    def __str__(self):
        # This method gives a nice way to display the ebook details.
        return f"Ebook: {self.title} by {self.author} | Genre: {self.genre} |
Price: {self.price} | Published: {self.publication_date}"


# Class 2 - Mystery
```

```python
# I created a Mystery class as a type of Ebook, with extra details specific to mystery books.
class Mystery(Ebook):
    def __init__(self, author, title, price, publication_date, average_length, mystery_level):
        # I used super() to inherit the Ebook properties and set the genre to "Mystery".
        super().__init__(author, title, genre="Mystery", price=price, publication_date=publication_date)
        self.average_length = average_length  # The average length of the mystery book
        self.mystery_level = mystery_level  # The level of mystery in the book

    def __str__(self):
        # This method adds specific details to the ebook display for mystery books.
        return super().__str__() + f" - Length: {self.average_length} pages, Mystery Level: {self.mystery_level}"


# Class 3 - Drama
# I created a Drama class as a type of Ebook, with extra details for drama books.
class Drama(Ebook):
    def __init__(self, author, title, price, publication_date, dramatic_tension):
        # I used super() to inherit the Ebook properties and set the genre to "Drama".
        super().__init__(author, title, genre="Drama", price=price, publication_date=publication_date)
        self.dramatic_tension = dramatic_tension  # The level of dramatic tension in the book

    def __str__(self):
        # This method adds specific details to the ebook display for drama books.
        return super().__str__() + f" | Dramatic Tension: {self.dramatic_tension}"


# Class 4 - Catalog
# I created a Catalog class to hold a collection of ebooks.
class Catalog:
    def __init__(self, catalog_id, name):
        # I used these attributes to manage the catalog.
        self.catalog_id = catalog_id  # The unique ID of the catalog
        self.name = name  # The name of the catalog
        self.total_ebooks = 0  # The total count of ebooks in the catalog
        self.ebooks = []  # A list to store all ebooks in the catalog

    def add_ebook(self, ebook):
        # I used this method to add an ebook to the catalog.
        self.ebooks.append(ebook)
```

```
        self.total_ebooks += 1

    def remove_ebook(self, ebook):
        # I used this method to remove an ebook from the catalog.
        if ebook in self.ebooks:
            self.ebooks.remove(ebook)
            self.total_ebooks -= 1

    def __str__(self):
        # This method displays the catalog's details and all ebooks in it.
        return f"Catalog: {self.name} (ID: {self.catalog_id}) | Total Ebooks:
{self.total_ebooks}\n" + "\n".join(str(ebook) for ebook in self.ebooks)


# Class 5 - Person
# I created a Person class to store personal information like name and email.
class Person:
    def __init__(self, name, email):
        self.name = name  # The name of the person
        self.email = email  # The email of the person

    def __str__(self):
        # This method formats the person's information for display.
        return f"Person: {self.name} | Email: {self.email}"


# Class 6 - Account
# I created an Account class to store account details related to a customer.
class Account:
    def __init__(self, customer_id, username, password):
        self.customer_id = customer_id  # The customer ID linked to this
account
        self.username = username  # The username of the account
        self.password = password  # The password for the account

    def __str__(self):
        # This method formats the account details for display, including the
password.
        return f"Account Username: {self.username} | Customer ID:
{self.customer_id} | Password: {self.password}"


# Class 7 - Customer
# I created a Customer class to manage customer information and link it to an
account.
class Customer:
    def __init__(self, name, contact_number, email, loyalty_member=False):
        self.name = name  # The customer's name
        self.contact_number = contact_number  # The customer's contact number
        self.email = email  # The customer's email address
        self.loyalty_member = loyalty_member  # Whether the customer is a
loyalty member
```

```
    def set_account(self, account):
        # I used this method to link an account to the customer.
        self.account = account

    def __str__(self):
        # This method formats the customer details, including loyalty status.
        loyalty_status = "Yes" if self.loyalty_member else "No"
        return f"Customer: {self.name} | Contact: {self.contact_number} |
Email: {self.email} | Loyalty Member: {loyalty_status}"


# Class 8 - ShoppingCart
# I created a ShoppingCart class to handle items added by a customer for
purchase.
class ShoppingCart:
    def __init__(self, cart_id):
        self.cart_id = cart_id  # The unique ID of the shopping cart
        self.items = []  # A list to store items in the cart
        self.total_items = 0  # The total count of items in the cart

    def add_item(self, item):
        # I used this method to add an item to the cart.
        self.items.append(item)
        self.total_items += 1

    def remove_item(self, item):
        # I used this method to remove an item from the cart.
        if item in self.items:
            self.items.remove(item)
            self.total_items -= 1

    def calculate_total_price(self):
        # This method calculates the total price of all items in the cart.
        return sum(item.price for item in self.items)

    def __str__(self):
        # This method displays the cart's details and all items in it.
        return f"ShoppingCart ID: {self.cart_id} | Total Items:
{self.total_items}\n" + "\n".join(str(item) for item in self.items)


# Class 9 - Order
# I created an Order class to store order details like order date and total
price.
class Order:
    def __init__(self, order_id, order_date, total_price):
        self.order_id = order_id  # The unique ID of the order
        self.order_date = order_date  # The date when the order was placed
        self.total_price = total_price  # The total price of the order

    def __str__(self):
        # This method formats the order details for display.
```

```
        return f"Order ID: {self.order_id} | Order Date: {self.order_date} |
Total Price: {self.total_price}"


# Class 10 - Discount
# I created a Discount class to apply a discount to an amount based on a
discount rate or amount.
class Discount:
    def __init__(self, discount_type, discount_id, values_amount):
        self.discount_type = discount_type  # The type of discount (e.g.,
loyalty program)
        self.discount_id = discount_id  # The unique ID of the discount
        self.values_amount = values_amount  # The value of the discount

    def apply_discount(self, amount):
        # This method applies the discount to a given amount.
        return amount * (1 - self.values_amount) if self.values_amount < 1
else amount - self.values_amount

    def __str__(self):
        # This method displays discount details.
        return f"Discount ID: {self.discount_id} | Type: {self.discount_type}
| Value Amount: {self.values_amount}"


# Class 11 - Loyalty
# I created a Loyalty class to store loyalty program details like discount
rate, membership duration, and loyalty status.
class Loyalty:
    def __init__(self, loyalty_discount_rate, membership_duration,
loyalty_member):
        self.loyalty_discount_rate = loyalty_discount_rate  # The discount
rate for loyalty members
        self.membership_duration = membership_duration  # The duration of the
loyalty membership in months
        self.loyalty_member = loyalty_member  # Whether the member is part of
the loyalty program

    def __str__(self):
        # This method formats the loyalty program details for display.
        loyalty_status = "Yes" if self.loyalty_member else "No"
        return f"Loyalty Discount Rate: {self.loyalty_discount_rate * 100}% |
Membership Duration: {self.membership_duration} months | Loyalty Member:
{loyalty_status}"


# Class 12 - Vat
# I created a Vat class to calculate VAT based on a given rate.
class Vat:
    def __init__(self, rateNumber):
        self.rateNumber = rateNumber  # The VAT rate number

    def calculate_vat(self, amount):
```

```
        # This method calculates the VAT on a given amount.
        return amount * self.rateNumber

    def __str__(self):
        # This method displays the VAT rate.
        return f"VAT Rate Number: {self.rateNumber * 100}%"


# Class 13 - Invoice
# I created an Invoice class to manage billing details, including tax and
total amount.
class Invoice:
    def __init__(self, tax, invoice_id, status, total_amount):
        self.tax = tax  # The tax percentage applied to the invoice
        self.invoice_id = invoice_id  # The unique ID of the invoice
        self.status = status  # The status of the invoice (e.g., paid or
pending)
        self.total_amount = total_amount  # The total amount due on the
invoice

    def __str__(self):
        # This method formats the invoice details for display.
        return f"Invoice ID: {self.invoice_id} | Tax: {self.tax}% | Status:
{self.status} | Total Amount: {self.total_amount}"


# Class 14 - Payment
# I created a Payment class to handle payment details like payment method and
status.
class Payment:
    def __init__(self, paymentID, paymentAmount, paymentMethod, status):
        self.paymentID = paymentID  # The unique ID of the payment
        self.paymentAmount = paymentAmount  # The total payment amount
        self.paymentMethod = paymentMethod  # The method of payment used
        self.status = status  # The payment status (e.g., complete or pending)

    def is_payment_complete(self):
        # This method checks if the payment status is marked as "complete".
        return self.status.lower() == "complete"

    def __str__(self):
        # This method formats the payment details for display.
        return f"Payment ID: {self.paymentID} | Method: {self.paymentMethod} |
Status: {self.status}"
```

5) Objects Code:

```
# Import each class individually
# I imported each class from my 'Class' module to create objects for each one
from Class import Ebook
from Class import Mystery
from Class import Drama
```

```python
from Class import Catalog
from Class import Customer
from Class import ShoppingCart
from Class import Order
from Class import Discount
from Class import Loyalty
from Class import Account
from Class import Payment
from Class import Vat
from Class import Person
from Class import Invoice


# Object 1 - Ebook
# I created an Ebook object with details like author, title, genre, price,
and publication date
ebook1 = Ebook(author="Harper Lee", title="To Kill a Mockingbird",
genre="Fiction", price=12.99, publication_date="1960")
print("Object 1 - Ebook:")
print(ebook1)

# Object 2 - Mystery
# I created a Mystery object, a type of Ebook, with additional details for
average length and mystery level
mystery_book = Mystery(author="Agatha Christie", title="The Murder of Roger
Ackroyd", price=30.0, publication_date="1926-06-30", average_length=320,
mystery_level="High")
print("Object 2 - Mystery:")
print(mystery_book)

# Object 3 - Drama
# I created a Drama object, another type of Ebook, with details about
dramatic tension
drama_book = Drama(author="Arthur Miller", title="Death of a Salesman",
price=15.0, publication_date="1949-02-10", dramatic_tension="High")
print("Object 3 - Drama:")
print(drama_book)

# Object 4 - Catalog
# I created a Catalog object and added multiple ebooks to it using
add_ebook()
catalog = Catalog(catalog_id="CAT123", name="Literature Classics")
catalog.add_ebook(ebook1)
catalog.add_ebook(mystery_book)
catalog.add_ebook(drama_book)
print("Object 4 - Catalog:")
print("Catalog contents:")
print(catalog)

# Object 5 - Person
# I created a Person object to store basic information like name and email
person_hamda = Person(name="Hamda Sami", email="hamda.sami@gmail.com")
print("Object 5 - Person:")
```

```
print(person_hamda)

# Object 6 - Account
# I created an Account object to store account details for a customer,
including username and password
account_hamda = Account(customer_id="CUST001", username="hamda.sami",
password="hamda423")
print("Object 6 - Account:")
print(account_hamda)


# Object 7 - Customer
# I created a Customer object to store customer details and linked it to an
account
customer_hamda = Customer(name="Hamda Sami", contact_number="0501234567",
email="hamda.sami@gmail.com", loyalty_member=True)
customer_hamda.set_account(account_hamda)  # I linked the customer to the
account
print("Object 7 - Customer:")
print(customer_hamda)

# Object 8 - ShoppingCart
# I created a ShoppingCart object and added items to it
shopping_cart = ShoppingCart(cart_id="CART001")
shopping_cart.add_item(ebook1)  # I added the Ebook item to the cart
shopping_cart.add_item(mystery_book)  # I added the Mystery item to the cart
print("Object 8 - ShoppingCart:")
print("Shopping Cart:")
print(shopping_cart)
print("Total Price: DHS", shopping_cart.calculate_total_price())  # I
calculated the total price of items in the cart

# Object 9 - Order
# I created an Order object using the total price from the shopping cart
order = Order(order_id="ORD001", order_date="2024-11-06",
total_price=shopping_cart.calculate_total_price())
print("Object 9 - Order:")
print(order)

# Object 10 - Discount
# I created a Discount object to apply a loyalty discount to the order's
total price
loyalty_discount = Discount(discount_type="Loyalty Program",
discount_id="DISC123", values_amount=0.1)
discounted_total = loyalty_discount.apply_discount(order.total_price)  # I
applied the discount to the total price
print("Object 10 - Discount:")
print(loyalty_discount)
print("Discounted Amount:", discounted_total)

# Object 11 - Loyalty
# I created a Loyalty object to represent a loyalty program with a discount
rate, membership duration, and loyalty status.
```

```
loyalty_program = Loyalty(loyalty_discount_rate=0.1, membership_duration=12,
loyalty_member=True)
print("Object 11 - Loyalty:")
print(loyalty_program)


# Object 12 - Vat
# I created a Vat object to calculate VAT on the discounted amount
vat_rate = Vat(rateNumber=0.05)
vat_amount = vat_rate.calculate_vat(discounted_total)  # I calculated VAT on
the discounted amount
print("Object 12 - Vat:")
print(vat_rate)
print("Calculated VAT Amount:", vat_amount)

# Object 13 - Invoice
# I created an Invoice object to store billing details, including tax,
status, and the total amount with VAT
invoice_total = discounted_total + vat_amount
invoice = Invoice(invoice_id="INV001", tax=5.0, status="Paid",
total_amount=invoice_total)
print("Object 13 - Invoice:")
print(invoice)

# Object 14 - Payment
# I created a Payment object to manage payment details, such as payment
method and status
payment = Payment(paymentID="PAY001", paymentAmount=invoice.total_amount,
paymentMethod="Credit Card", status="Complete")
print("Object 14 - Payment:")
print(payment)
print("Payment Complete:", payment.is_payment_complete())  # I checked if the
payment status is complete
```

6) The Outputs for each class:

```
# Class 1 - Ebook
# I created an Ebook class to store details about ebooks.
class Ebook:
    def __init__(self, author, title, genre, price, publication_date):
        # I used these attributes to hold the basic information of an ebook.
        self.author = author  # The author of the ebook
        self.title = title  # The title of the ebook
        self.genre = genre  # The genre of the ebook
        self.price = price  # The price of the ebook
        self.publication_date = publication_date  # The publication date of
the ebook

    def __str__(self):
        # This method gives a nice way to display the ebook details.
```

```
        return f"Ebook: {self.title} by {self.author} | Genre: {self.genre} |
Price: {self.price} | Published: {self.publication_date}"

from Class import Ebook
# Object 1 - Ebook
# I created an Ebook object with details like author, title, genre, price,
and publication date
ebook1 = Ebook(author="Harper Lee", title="To Kill a Mockingbird",
genre="Fiction", price=12.99, publication_date="1960")
print("Object 1 - Ebook:")
print(ebook1)
```

Output:

Object 1 - Ebook:
Ebook: To Kill a Mockingbird by Harper Lee | Genre: Fiction | Price: 12.99 | Published: 1960

```
# Class 2 - Mystery
# I created a Mystery class as a type of Ebook, with extra details specific
to mystery books.
class Mystery(Ebook):
    def __init__(self, author, title, price, publication_date, average_length,
mystery_level):
        # I used super() to inherit the Ebook properties and set the genre to
"Mystery".
        super().__init__(author, title, genre="Mystery", price=price,
publication_date=publication_date)
        self.average_length = average_length  # The average length of the
mystery book
        self.mystery_level = mystery_level  # The level of mystery in the book

    def __str__(self):
        # This method adds specific details to the ebook display for mystery
books.
        return super().__str__() + f" - Length: {self.average_length} pages,
Mystery Level: {self.mystery_level}"
from Class import Mystery
# Object 2 - Mystery
# I created a Mystery object, a type of Ebook, with additional details for
average length and mystery level
mystery_book = Mystery(author="Agatha Christie", title="The Murder of Roger
Ackroyd", price=30.0, publication_date="1926-06-30", average_length=320,
mystery_level="High")
print("Object 2 - Mystery:")
print(mystery_book)
```

Output:

Object 2 - Mystery:

Ebook: The Murder of Roger Ackroyd by Agatha Christie | Genre: Mystery | Price: 30.0 | Published: 1926-06-30 - Length: 320 pages, Mystery Level: High

```
# Class 3 - Drama
# I created a Drama class as a type of Ebook, with extra details for drama
books.
class Drama(Ebook):
    def __init__(self, author, title, price, publication_date,
dramatic_tension):
        # I used super() to inherit the Ebook properties and set the genre to
"Drama".
        super().__init__(author, title, genre="Drama", price=price,
publication_date=publication_date)
        self.dramatic_tension = dramatic_tension  # The level of dramatic
tension in the book

    def __str__(self):
        # This method adds specific details to the ebook display for drama
books.
        return super().__str__() + f" | Dramatic Tension:
{self.dramatic_tension}"
from Class import Drama
# Object 3 - Drama
# I created a Drama object, another type of Ebook, with details about
dramatic tension
drama_book = Drama(author="Arthur Miller", title="Death of a Salesman",
price=15.0, publication_date="1949-02-10", dramatic_tension="High")
print("Object 3 - Drama:")
print(drama_book)
```

Output:

Object 3 - Drama:
Ebook: Death of a Salesman by Arthur Miller | Genre: Drama | Price: 15.0 | Published: 1949-02-10 | Dramatic Tension: High

```
# Class 4 - Catalog
# I created a Catalog class to hold a collection of ebooks.
class Catalog:
    def __init__(self, catalog_id, name):
        # I used these attributes to manage the catalog.
        self.catalog_id = catalog_id  # The unique ID of the catalog
        self.name = name  # The name of the catalog
        self.total_ebooks = 0  # The total count of ebooks in the catalog
        self.ebooks = []  # A list to store all ebooks in the catalog
```

```
    def add_ebook(self, ebook):
        # I used this method to add an ebook to the catalog.
        self.ebooks.append(ebook)
        self.total_ebooks += 1

    def remove_ebook(self, ebook):
        # I used this method to remove an ebook from the catalog.
        if ebook in self.ebooks:
            self.ebooks.remove(ebook)
            self.total_ebooks -= 1

    def __str__(self):
        # This method displays the catalog's details and all ebooks in it.
        return f"Catalog: {self.name} (ID: {self.catalog_id}) | Total Ebooks:
{self.total_ebooks}\n" + "\n".join(str(ebook) for ebook in self.ebooks)
from Class import Catalog
# Object 4 - Catalog
# I created a Catalog object and added multiple ebooks to it using
add_ebook()
catalog = Catalog(catalog_id="CAT123", name="Literature Classics")
catalog.add_ebook(ebook1)
catalog.add_ebook(mystery_book)
catalog.add_ebook(drama_book)
print("Object 4 - Catalog:")
print("Catalog contents:")
print(catalog)
```

Output:

Object 4 - Catalog:
Catalog contents:
Catalog: Literature Classics (ID: CAT123) | Total Ebooks: 3
Ebook: To Kill a Mockingbird by Harper Lee | Genre: Fiction | Price: 12.99 | Published: 1960
Ebook: The Murder of Roger Ackroyd by Agatha Christie | Genre: Mystery | Price: 30.0 |
Published: 1926-06-30 - Length: 320 pages, Mystery Level: High
Ebook: Death of a Salesman by Arthur Miller | Genre: Drama | Price: 15.0 | Published:
1949-02-10 | Dramatic Tension: High

```
# Class 5 - Person
# I created a Person class to store personal information like name and email.
class Person:
    def __init__(self, name, email):
        self.name = name  # The name of the person
        self.email = email  # The email of the person

    def __str__(self):
        # This method formats the person's information for display.
        return f"Person: {self.name} | Email: {self.email}"
```

```
from Class import Person
# Object 5 - Person
# I created a Person object to store basic information like name and email
person_hamda = Person(name="Hamda Sami", email="hamda.sami@gmail.com")
print("Object 5 - Person:")
print(person_hamda)
```

Output:

Object 5 - Person:
Person: Hamda Sami | Email: hamda.sami@gmail.com

```
# Class 6 - Account
# I created an Account class to store account details related to a customer.
class Account:
  def __init__(self, customer_id, username, password):
      self.customer_id = customer_id  # The customer ID linked to this
account
      self.username = username  # The username of the account
      self.password = password  # The password for the account

  def __str__(self):
      # This method formats the account details for display, including the
password.
      return f"Account Username: {self.username} | Customer ID:
{self.customer_id} | Password: {self.password}"
from Class import Account
# Object 6 - Account
# I created an Account object to store account details for a customer,
including username and password
account_hamda = Account(customer_id="CUST001", username="hamda.sami",
password="hamda423")
print("Object 6 - Account:")
print(account_hamda)
```

Output:

Object 6 - Account:
Account Username: hamda.sami | Customer ID: CUST001| Password: hamda423

```
# Class 7 - Customer
# I created a Customer class to manage customer information and link it to an
account.
class Customer:
   def __init__(self, name, contact_number, email, loyalty_member=False):
       self.name = name  # The customer's name
       self.contact_number = contact_number  # The customer's contact number
       self.email = email  # The customer's email address
```

```
        self.loyalty_member = loyalty_member  # Whether the customer is a
loyalty member

    def set_account(self, account):
        # I used this method to link an account to the customer.
        self.account = account

    def __str__(self):
        # This method formats the customer details, including loyalty status.
        loyalty_status = "Yes" if self.loyalty_member else "No"
        return f"Customer: {self.name} | Contact: {self.contact_number} |
Email: {self.email} | Loyalty Member: {loyalty_status}"

from Class import Customer
# Object 7 - Customer
# I created a Customer object to store customer details and linked it to an
account
customer_hamda = Customer(name="Hamda Sami", contact_number="0501234567",
email="hamda.sami@gmail.com", loyalty_member=True)
customer_hamda.set_account(account_hamda)  # I linked the customer to the
account
print("Object 7 - Customer:")
print(customer_hamda)
```

Output:

Object 7 - Customer:
Customer: Hamda Sami | Contact: 0501234567 | Email: hamda.sami@gmail.com | Loyalty Member: Yes

```
# Class 8 - ShoppingCart
# I created a ShoppingCart class to handle items added by a customer for
purchase.
class ShoppingCart:
    def __init__(self, cart_id):
        self.cart_id = cart_id  # The unique ID of the shopping cart
        self.items = []  # A list to store items in the cart
        self.total_items = 0  # The total count of items in the cart

    def add_item(self, item):
        # I used this method to add an item to the cart.
        self.items.append(item)
        self.total_items += 1

    def remove_item(self, item):
        # I used this method to remove an item from the cart.
        if item in self.items:
            self.items.remove(item)
            self.total_items -= 1
```

```
    def calculate_total_price(self):
        # This method calculates the total price of all items in the cart.
        return sum(item.price for item in self.items)

    def __str__(self):
        # This method displays the cart's details and all items in it.
        return f"ShoppingCart ID: {self.cart_id} | Total Items:
{self.total_items}\n" + "\n".join(str(item) for item in self.items)
from Class import ShoppingCart
# Object 8 - ShoppingCart
# I created a ShoppingCart object and added items to it
shopping_cart = ShoppingCart(cart_id="CART001")
shopping_cart.add_item(ebook1)  # I added the Ebook item to the cart
shopping_cart.add_item(mystery_book)  # I added the Mystery item to the cart
print("Object 8 - ShoppingCart:")
print("Shopping Cart:")
print(shopping_cart)
print("Total Price: DHS", shopping_cart.calculate_total_price())  # I
calculated the total price of items in the cart
```

Output:

Object 8 - ShoppingCart:
Shopping Cart:
ShoppingCart ID: CART001 | Total Items: 2
Ebook: To Kill a Mockingbird by Harper Lee | Genre: Fiction | Price: 12.99 | Published: 1960
Ebook: The Murder of Roger Ackroyd by Agatha Christie | Genre: Mystery | Price: 30.0 |
Published: 1926-06-30 - Length: 320 pages, Mystery Level: High
Total Price: DHS 42.99

```
# Class 9 - Order
# I created an Order class to store order details like order date and total
price.
class Order:
    def __init__(self, order_id, order_date, total_price):
        self.order_id = order_id  # The unique ID of the order
        self.order_date = order_date  # The date when the order was placed
        self.total_price = total_price  # The total price of the order

    def __str__(self):
        # This method formats the order details for display.
        return f"Order ID: {self.order_id} | Order Date: {self.order_date} |
Total Price: {self.total_price}"

from Class import Order
# Object 9 - Order
# I created an Order object using the total price from the shopping cart
```

```
order = Order(order_id="ORD001", order_date="2024-11-06",
total_price=shopping_cart.calculate_total_price())
print("Object 9 - Order:")
print(order)
```

Output:

Object 9 - Order:
Order ID: ORD001 | Order Date: 2024-11-06 | Total Price: 42.99

```
# Class 10 - Discount
# I created a Discount class to apply a discount to an amount based on a
discount rate or amount.
class Discount:
    def __init__(self, discount_type, discount_id, values_amount):
        self.discount_type = discount_type  # The type of discount (e.g.,
loyalty program)
        self.discount_id = discount_id  # The unique ID of the discount
        self.values_amount = values_amount  # The value of the discount

    def apply_discount(self, amount):
        # This method applies the discount to a given amount.
        return amount * (1 - self.values_amount) if self.values_amount < 1
else amount - self.values_amount

    def __str__(self):
        # This method displays discount details.
        return f"Discount ID: {self.discount_id} | Type: {self.discount_type}
| Value Amount: {self.values_amount}"
from Class import Discount
# Object 10 - Discount
# I created a Discount object to apply a loyalty discount to the order's
total price
loyalty_discount = Discount(discount_type="Loyalty Program",
discount_id="DISC123", values_amount=0.1)
discounted_total = loyalty_discount.apply_discount(order.total_price)  # I
applied the discount to the total price
print("Object 10 - Discount:")
print(loyalty_discount)
print("Discounted Amount:", discounted_total)
```

Output:

Object 10 - Discount:
Discount ID: DISC123 | Type: Loyalty Program | Value Amount: 0.1
Discounted Amount: 38.691

```
# Class 11 - Loyalty
```

```
# I created a Loyalty class to store loyalty program details like discount
rate, membership duration, and loyalty status.
class Loyalty:
   def __init__(self, loyalty_discount_rate, membership_duration,
loyalty_member):
       self.loyalty_discount_rate = loyalty_discount_rate  # The discount
rate for loyalty members
       self.membership_duration = membership_duration  # The duration of the
loyalty membership in months
       self.loyalty_member = loyalty_member  # Whether the member is part of
the loyalty program

   def __str__(self):
       # This method formats the loyalty program details for display.
       loyalty_status = "Yes" if self.loyalty_member else "No"
       return f"Loyalty Discount Rate: {self.loyalty_discount_rate * 100}% |
Membership Duration: {self.membership_duration} months | Loyalty Member:
{loyalty_status}"

from Class import Loyalty
# Object 11 - Loyalty
# I created a Loyalty object to represent a loyalty program with a discount
rate, membership duration, and loyalty status.
loyalty_program = Loyalty(loyalty_discount_rate=0.1, membership_duration=12,
loyalty_member=True)
print("Object 11 - Loyalty:")
print(loyalty_program)
```

Output:

Object 11 - Loyalty:
Loyalty Discount Rate: 10.0% | Membership Duration: 12 months | Loyalty Member: Yes

```
# Class 12 - Vat
# I created a Vat class to calculate VAT based on a given rate.
class Vat:
   def __init__(self, rateNumber):
       self.rateNumber = rateNumber  # The VAT rate number

   def calculate_vat(self, amount):
       # This method calculates the VAT on a given amount.
       return amount * self.rateNumber

   def __str__(self):
       # This method displays the VAT rate.
       return f"VAT Rate Number: {self.rateNumber * 100}%"

from Class import Vat
```

```
# Object 12 - Vat
# I created a Vat object to calculate VAT on the discounted amount
vat_rate = Vat(rateNumber=0.05)
vat_amount = vat_rate.calculate_vat(discounted_total)  # I calculated VAT on
the discounted amount
print("Object 12 - Vat:")
print(vat_rate)
print("Calculated VAT Amount:", vat_amount)
```

Output:

Object 12 - Vat:
VAT Rate Number: 5.0%
Calculated VAT Amount: 1.9345500000000002

```
# Class 13 - Invoice
# I created an Invoice class to manage billing details, including tax and
total amount.
class Invoice:
    def __init__(self, tax, invoice_id, status, total_amount):
        self.tax = tax  # The tax percentage applied to the invoice
        self.invoice_id = invoice_id  # The unique ID of the invoice
        self.status = status  # The status of the invoice (e.g., paid or
pending)
        self.total_amount = total_amount  # The total amount due on the
invoice

    def __str__(self):
        # This method formats the invoice details for display.
        return f"Invoice ID: {self.invoice_id} | Tax: {self.tax}% | Status:
{self.status} | Total Amount: {self.total_amount}"
from Class import Invoice
# Object 13 - Invoice
# I created an Invoice object to store billing details, including tax,
status, and the total amount with VAT
invoice_total = discounted_total + vat_amount
invoice = Invoice(invoice_id="INV001", tax=5.0, status="Paid",
total_amount=invoice_total)
print("Object 13 - Invoice:")
print(invoice)
```

Output:

Object 13 - Invoice:
Invoice ID: INV001 | Tax: 5.0% | Status: Paid | Total Amount: 40.625550000000004

```
# Class 14 - Payment
# I created a Payment class to handle payment details like payment method and
status.
class Payment:
    def __init__(self, paymentID, paymentAmount, paymentMethod, status):
        self.paymentID = paymentID  # The unique ID of the payment
        self.paymentAmount = paymentAmount  # The total payment amount
        self.paymentMethod = paymentMethod  # The method of payment used
        self.status = status  # The payment status (e.g., complete or pending)

    def is_payment_complete(self):
        # This method checks if the payment status is marked as "complete".
        return self.status.lower() == "complete"

    def __str__(self):
        # This method formats the payment details for display.
        return f"Payment ID: {self.paymentID} | Method: {self.paymentMethod} |
Status: {self.status}"
from Class import Payment
# Object 14 - Payment
# I created a Payment object to manage payment details, such as payment
method and status
payment = Payment(paymentID="PAY001", paymentAmount=invoice.total_amount,
paymentMethod="Credit Card", status="Complete")
print("Object 14 - Payment:")
print(payment)
print("Payment Complete:", payment.is_payment_complete())  # I checked if the
payment status is complete
```

Output:

Object 14 - Payment:
Payment ID: PAY001 | Method: Credit Card | Status: Complete
Payment Complete: True

Summary:

In this assignment, I learned how to use UML diagrams to design and visually organize a system before implementing it in Python code. First, through UML diagrams, I represented classes with attribute boxes, showing key properties like name, email, or price. This made it clear to me what each class holds and helped me visualize the structure of the system. Additionally, I learned the differences between various types of arrows in UML diagrams, which represent relationships between classes. For example, I discovered that inheritance arrows indicate an "is a" relationship, where one class is a subtype of another, such as a Mystery being a type of Ebook. On the other hand, aggregation arrows show a "part of" relationship, where one class can contain another without being dependent on it, like a Catalog containing multiple Ebooks. Meanwhile, composition arrows represent a "has a"

relationship, where one class is an essential part of another; for instance, an Order has an Invoice, and if the main class is removed, the composed part is also removed. Furthermore, I used cardinality to indicate how many instances of one class can connect to instances of another, which gave me a better understanding of the strength and number of connections between classes. Overall, UML diagrams helped me achieve a clear, organized design, provided guidance throughout my coding process, and made it easier to build structured, manageable systems.

Diagram link:
https://app.diagrams.net/#G10Unr-lrVe66lVE1cwFYfihCPOv5Vwqzm#%7B%22pageId%22%3A%22gaGG9LkFRIwh85QHapJk%22%7D

GitHub line for the codes: https://github.com/hamdaSami/Assig-2