Inheritance:

When a class takes on attributes and methods from another class. It's like a child inheriting traits from a parent.

Association:

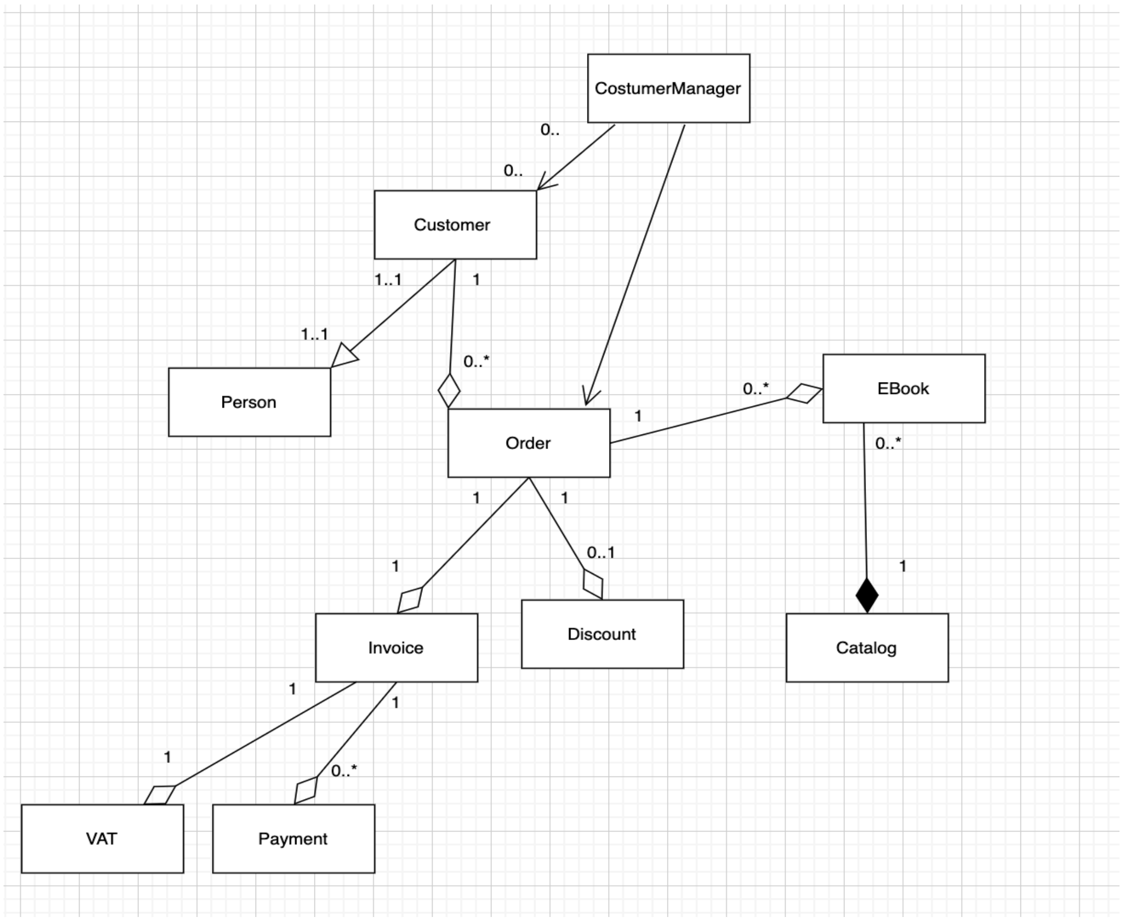
A general relationship between two classes where they can work together, but neither owns the other. Think of it as a partnership.

Aggregation:

A "has-a" relationship where one class contains another, but the contained class can exist independently. For example, a library has books, but books can exist without a library.

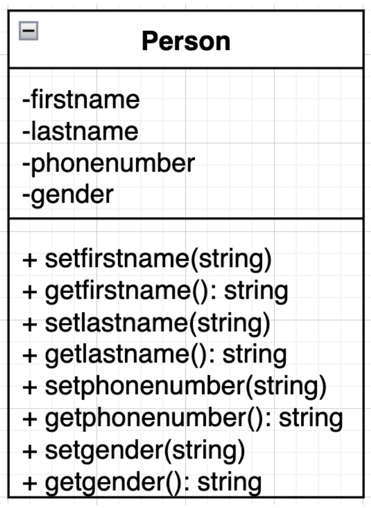
Composition:

A stronger "has-a" relationship where one class owns another, and if the owner is deleted, the contained object is also deleted. For example, a house is composed of rooms, and if the house is demolished, the rooms are too.



Person:

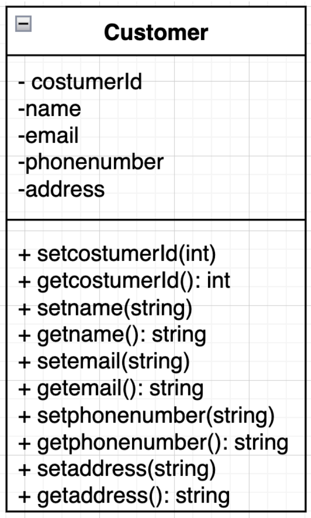
Relationship (Inheritance): The Customer class extends Person, as every customer is a person but with additional customer-specific details. This setup follows the principle of inheritance, allowing Customer to reuse attributes and methods of Person.



Customer:

Relationship (Association): CustomerManager has a one-to-many association with Customer, meaning it can manage multiple customers. This association is necessary for managing customer accounts.

Relationship (Association with Order): Each Customer can have multiple Orders, but each Order is associated with only one Customer. This helps track each customer's transactions.



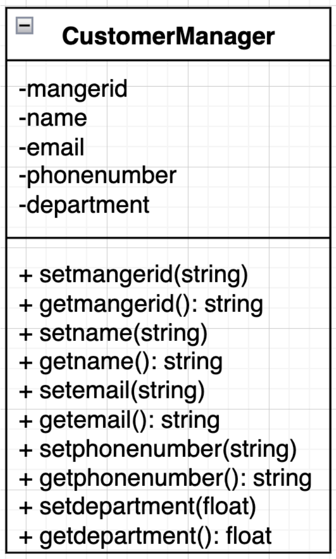
```
class Customer:

   def __init__(self, name, email):

       self.name = name

       self.email = email


   def __str__(self):

       return f'Customer: {self.name}, Email: {self.email}'

customer1 = Customer("Alice", "alice@example.com")

print(customer1)
```

Output:

```
Customer: Alice, Email: alice@example.com
```

CustomerManager:

Relationship (Association with Customer): CustomerManager has a one-to-many relationship with Customer, allowing it to manage multiple customer accounts.

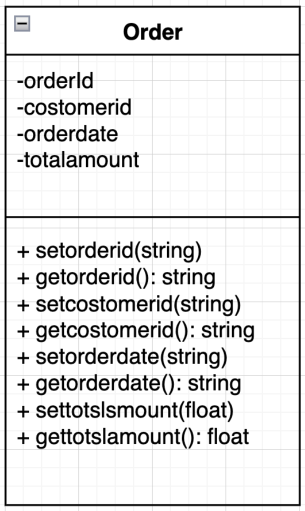


Order:

Relationship (Aggregation with EBook): Order aggregates EBook because an order can include multiple e-books without owning them directly; the e-books exist independently in the catalog.

Relationship (Association with Discount): Order can have an optional discount (0 or 1), allowing it to adjust the total based on the discount applied.

Relationship (Association with Invoice): Order and Invoice have a one-to-one relationship, where each order generates a single invoice to track payment.



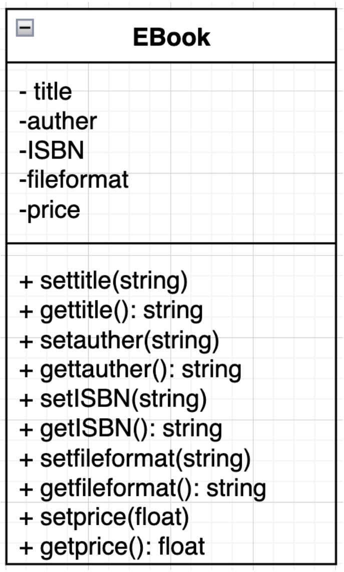
```python
class Order:
   def __init__(self, order_id, customer, ebook_list):
       self.order_id = order_id
       self.customer = customer
       self.ebook_list = ebook_list

   def __str__(self):
       ebook_titles = ", ".join([ebook.title for ebook in
self.ebook_list])
       return f'Order ID: {self.order_id}, Customer:
{self.customer.name}, eBooks: {ebook_titles}'
order1 = Order(101, customer1, [ebook1])
print(order1)
```

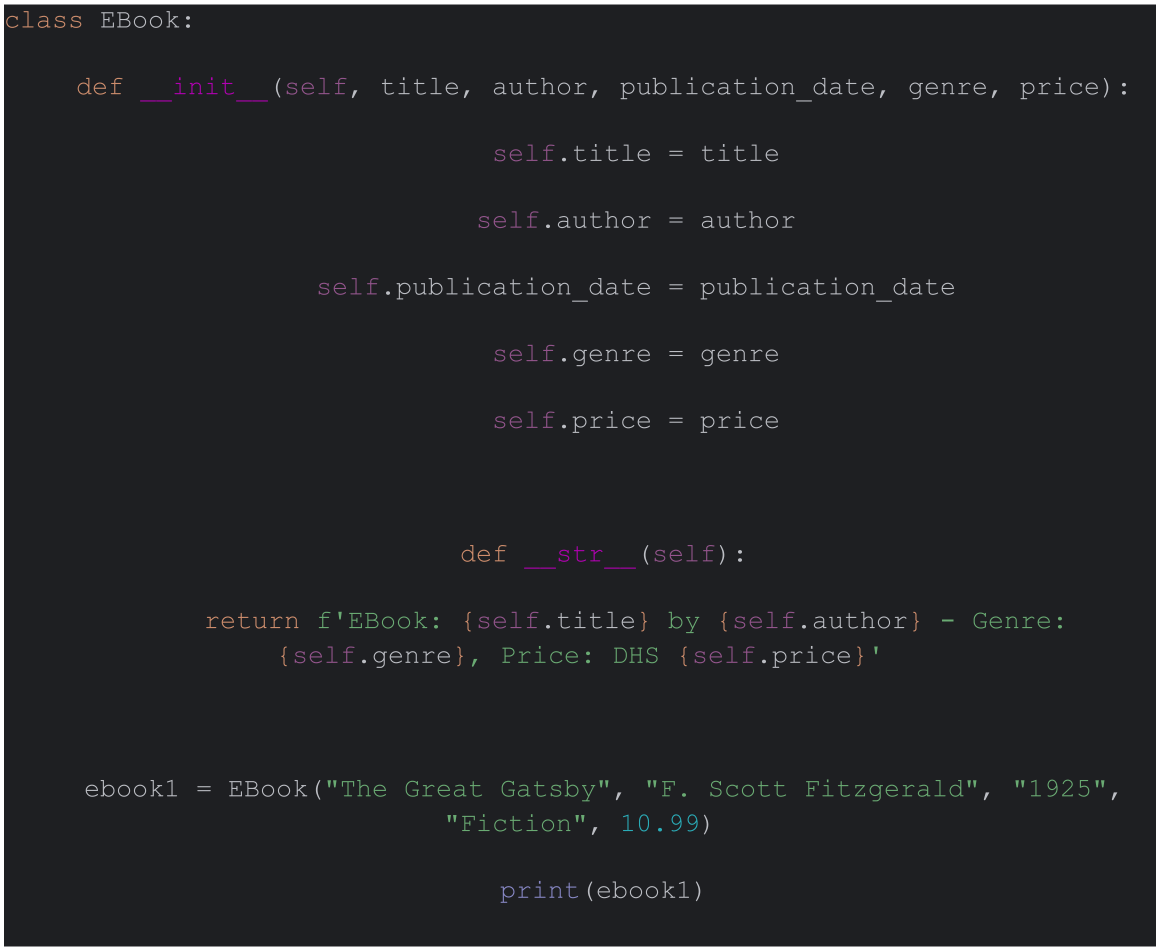
Output:

Order ID: 101, Customer: Alice, eBooks: The Great Gatsby

EBook:

Relationship (Composition with Catalog): The Catalog has a composition relationship with EBook, meaning that EBooks belong to the catalog. If a catalog is deleted, the e-books within it are also removed. This is a strong relationship because EBook instances are logically part of the catalog.





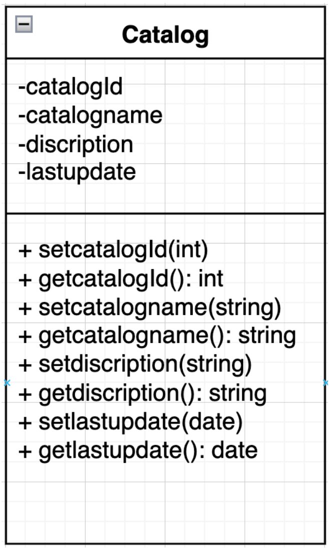
```
class EBook:

    def __init__(self, title, author, publication_date, genre, price):

        self.title = title

        self.author = author

        self.publication_date = publication_date

        self.genre = genre

        self.price = price


    def __str__(self):

        return f'EBook: {self.title} by {self.author} - Genre:
        {self.genre}, Price: DHS {self.price}'



ebook1 = EBook("The Great Gatsby", "F. Scott Fitzgerald", "1925",
"Fiction", 10.99)

print(ebook1)
```

Output:

EBook: The Great Gatsby by F. Scott Fitzgerald - Genre: Fiction, Price: DHS 10.99

Catalog:

Relationship (Composition with EBook): As described, Catalog and EBook are in a composition relationship, showing that e-books are tightly integrated within the catalog's lifecycle.



```
class Catalog:
   def __init__(self, ebooks):
       self.ebooks = ebooks

   def __str__(self):
       ebook_list = "\n".join([str(ebook) for ebook in self.ebooks])
       return f'Catalog:\n{ebook_list}'
catalog = Catalog([ebook1])
print(catalog)
```

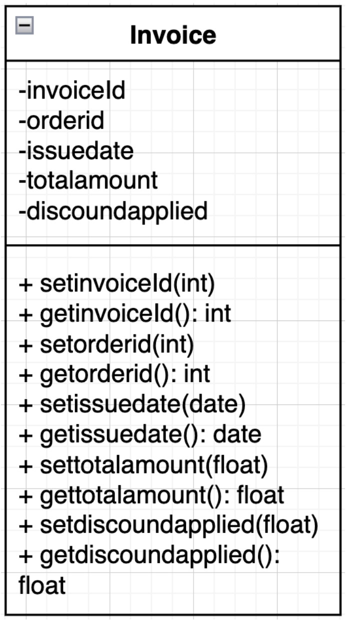
Output:

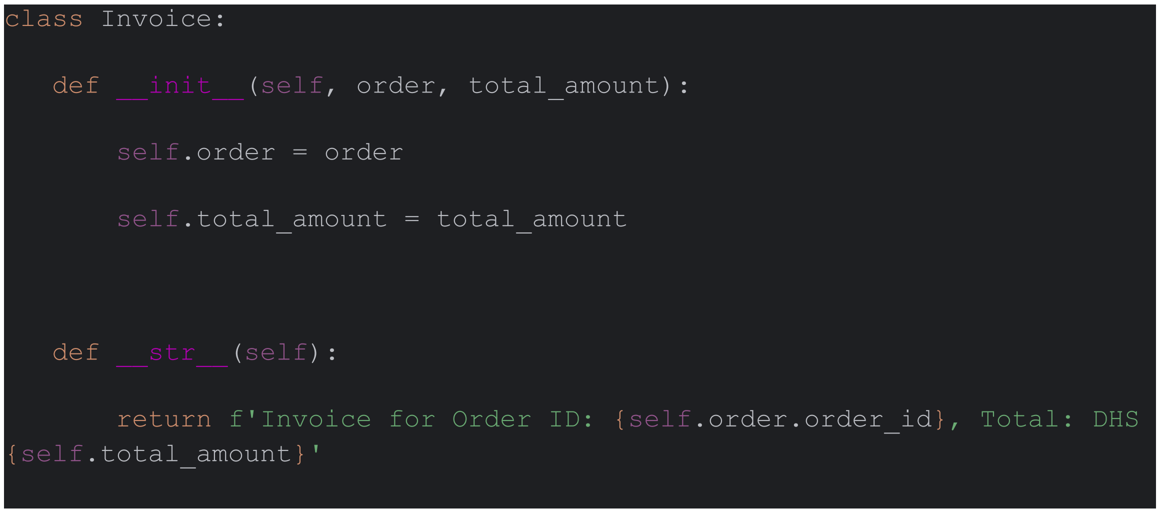
Catalog: EBook: The Great Gatsby by F. Scott Fitzgerald - Genre: Fiction, Price: DHS 10.99

Invoice:

Relationship (Aggregation with VAT): The Invoice aggregates VAT, showing that VAT is part of the invoice details but can exist independently as a tax object.

Relationship (Aggregation with Payment): Invoice has a one-to-many aggregation with Payment, allowing invoices to track multiple payment methods, like installments or partial payments.





```
class Invoice:

   def __init__(self, order, total_amount):

       self.order = order

       self.total_amount = total_amount



   def __str__(self):

       return f'Invoice for Order ID: {self.order.order_id}, Total: DHS {self.total_amount}'
```

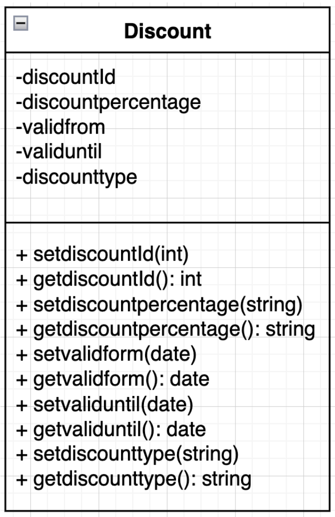
```
invoice = Invoice(order1, 10.99)

print(invoice)
```

Output:

```
Invoice for Order ID: 101, Total: DHS 10.99
```

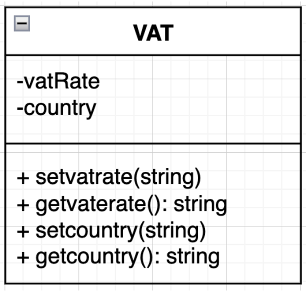
Discount:

Relationship (Association with Order): An Order can apply a single discount. This association is helpful for tracking and calculating the discounted amount in the final total.



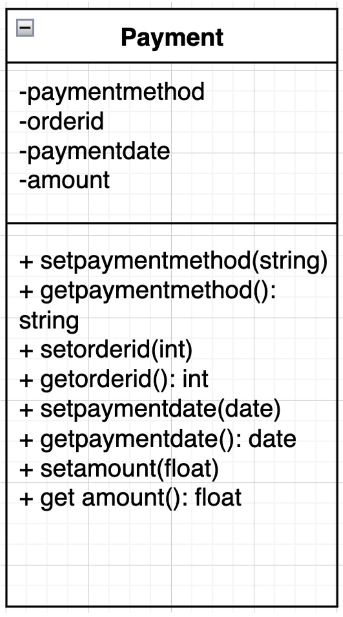
VAT:

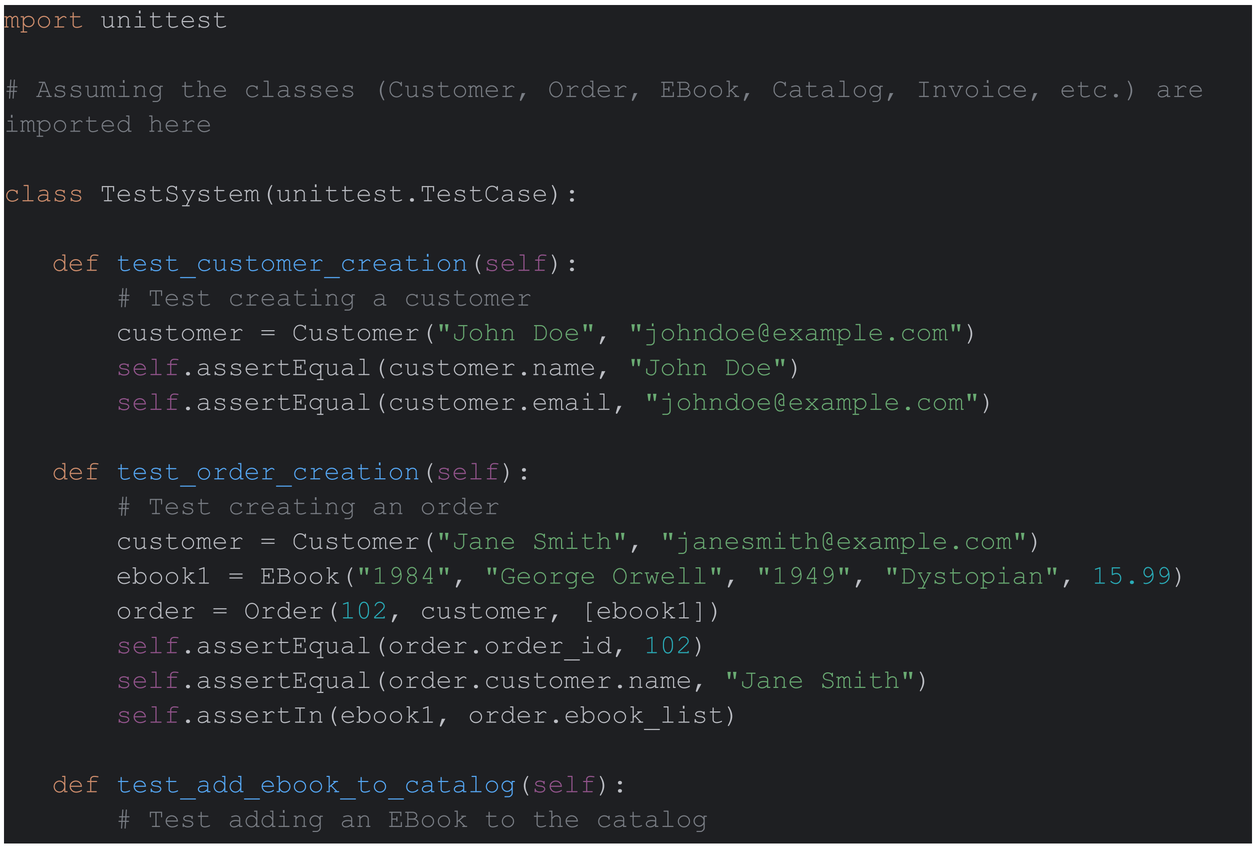
Relationship (Aggregation with Invoice): VAT is an optional part of the invoice but can be used independently across multiple invoices if required.



Payment:

Relationship (Aggregation with Invoice): Invoice can have multiple payments, allowing flexibility in the payment structure.





```
mport unittest

# Assuming the classes (Customer, Order, EBook, Catalog, Invoice, etc.) are
imported here

class TestSystem(unittest.TestCase):

   def test_customer_creation(self):
       # Test creating a customer
       customer = Customer("John Doe", "johndoe@example.com")
       self.assertEqual(customer.name, "John Doe")
       self.assertEqual(customer.email, "johndoe@example.com")

   def test_order_creation(self):
       # Test creating an order
       customer = Customer("Jane Smith", "janesmith@example.com")
       ebook1 = EBook("1984", "George Orwell", "1949", "Dystopian", 15.99)
       order = Order(102, customer, [ebook1])
       self.assertEqual(order.order_id, 102)
       self.assertEqual(order.customer.name, "Jane Smith")
       self.assertIn(ebook1, order.ebook_list)

   def test_add_ebook_to_catalog(self):
       # Test adding an EBook to the catalog
```

```python
        ebook2 = EBook("Brave New World", "Aldous Huxley", "1932", "Science
Fiction", 12.99)
        catalog = Catalog([ebook2])
        self.assertIn(ebook2, catalog.ebooks)

    def test_invoice_creation(self):
        # Test creating an invoice
        customer = Customer("Alice Johnson", "alice@example.com")
        ebook = EBook("The Catcher in the Rye", "J.D. Salinger", "1951",
"Fiction", 13.99)
        order = Order(103, customer, [ebook])
        invoice = Invoice(order, 13.99)
        self.assertEqual(invoice.total_amount, 13.99)
        self.assertEqual(invoice.order.order_id, 103)

    def test_discount_application(self):
        # Test applying a discount to an order
        customer = Customer("Bob Green", "bob@example.com")
        ebook = EBook("Fahrenheit 451", "Ray Bradbury", "1953", "Dystopian",
9.99)
        order = Order(104, customer, [ebook])
        discount = 2.00  # Let's say the discount is a fixed amount
        discounted_total = order.ebook_list[0].price - discount
        self.assertEqual(discounted_total, 7.99)

    def test_vat_aggregation_with_invoice(self):
        # Test VAT aggregation in an invoice
        customer = Customer("Charlie White", "charlie@example.com")
        ebook = EBook("To Kill a Mockingbird", "Harper Lee", "1960", "Fiction",
14.99)
        order = Order(105, customer, [ebook])
        vat = VAT(15)  # Assume 15% VAT
        total_with_vat = order.ebook_list[0].price + (order.ebook_list[0].price
* vat.get_rate() / 100)
        invoice = Invoice(order, total_with_vat)
        self.assertEqual(invoice.total_amount, total_with_vat)

    def test_ebook_removal_from_catalog(self):
        # Test removing an e-book from the catalog
        ebook1 = EBook("The Hobbit", "J.R.R. Tolkien", "1937", "Fantasy",
10.99)
        ebook2 = EBook("The Lord of the Rings", "J.R.R. Tolkien", "1954",
"Fantasy", 20.99)
        catalog = Catalog([ebook1, ebook2])
        catalog.ebooks.remove(ebook1)
        self.assertNotIn(ebook1, catalog.ebooks)

if __name__ == "__main__":
    unittest.main()
```

Object-oriented design (OOD) is about organizing code into classes, which are blueprints for creating objects. These classes group together data (attributes) and functions (methods) that work on that data. This design approach makes it easier to manage, update, and reuse code. Key principles in OOD include encapsulation, where we hide the internal details and only show what's necessary; abstraction, which simplifies complex systems; and inheritance, where one class can inherit behaviors from another, reducing code repetition.

In UML (Unified Modeling Language), relationships between classes are represented with different types of lines. Inheritance is shown with a line and a triangle, meaning one class is a type of another. Association shows a general relationship between two classes, like a "has-a" connection. Aggregation is a weaker form of association where one class contains another but can exist independently. Composition is a stronger form where one class depends on the other for its existence.

In Python, implementing these ideas involves defining classes with the class keyword. Each class can have an __init__ method to initialize the data it will hold. The self keyword is used to refer to the instance of the class. Relationships like inheritance are created by simply making one class inherit from another. For example, a Customer class can inherit from a Person class, allowing it to reuse and extend the behavior of Person. Aggregation and composition are modeled by including objects within a class, showing how different parts of the system work together.