Problem 2: Inventory Management System Optimization

Scenario:

You have been hired by a retail company to optimize their inventory management system. The company wants to minimize stockouts and overstock situations while maximizing inventory turnover and profitability.

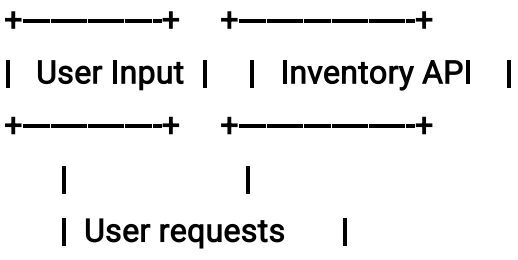
Tasks:

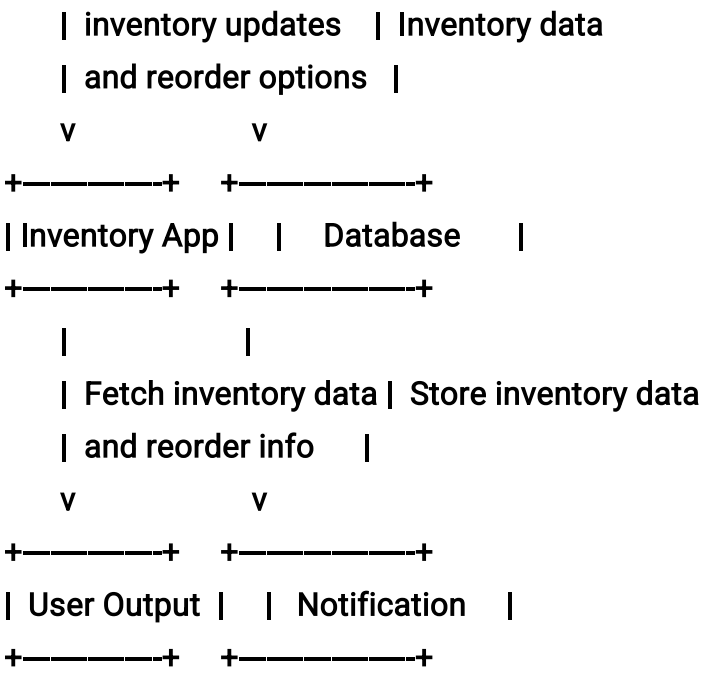
1. Model the inventory system: Define the structure of the inventory system, including products, warehouses, and current stock levels.
2. Implement an inventory tracking application: Develop a Python application that tracks inventory levels in real-time and alerts when stock levels fall below a certain threshold.
3. Optimize inventory ordering: Implement algorithms to calculate optimal reorder points and quantities based on historical sales data, lead times, and demand forecasts.
4. Generate reports: Provide reports on inventory turnover rates, stockout occurrences, and cost implications of overstock situations.
5. User interaction: Allow users to input product IDs or names to view current stock levels, reorder recommendations, and historical data.

Deliverables:

- **Data Flow Diagram**: Illustrate how data flows within the inventory management system, from input (e.g., sales data, inventory adjustments) to output (e.g., reorder alerts, reports).
- **Pseudocode and Implementation**: Provide pseudocode and actual code demonstrating how inventory levels are tracked, reorder points are calculated, and reports are generated.
- **Documentation**: Explain the algorithms used for reorder optimization, how historical data influences decisions, and any assumptions made (e.g., constant lead times).
- **User Interface**: Develop a user-friendly interface for accessing inventory information, viewing reports, and receiving alerts.
- **Assumptions and Improvements**: Discuss assumptions about demand patterns, supplier reliability, and potential improvements for the inventory management system's efficiency and accuracy.

Approach:





1. User Input: Users input product IDs, request inventory updates, and reorder options.
2. Inventory API: Provides real-time inventory data and alerts.
3. Inventory App: Central application that processes user requests and interacts with the database.
4. Database: Stores current stock levels, product details, and transaction history.
5. User Output: Displays current stock levels, reorder recommendations, and historical data to users.
6. Notification: Sends alerts for low stock levels or other important inventory updates.

Pseudocode:

```
# Define the inventory system structure
define Product, Warehouse, and InventoryLevel classes

# Implement the inventory tracking application
class InventoryTrackingApp:
    def __init__(self, products, warehouses):
        self.products = products
        self.warehouses = warehouses
        self.inventory_levels = self.initialize_inventory_levels()
```

```
    def initialize_inventory_levels(self):
        # Initialize inventory levels based on the provided data
        inventory_levels = []
        for product in self.products:
            for warehouse in self.warehouses:
                inventory_level = InventoryLevel(product, warehouse, initial_stock)
                inventory_levels.append(inventory_level)
        return inventory_levels

    def track_inventory(self):
        # Monitor inventory levels and send alerts when stock falls below threshold
        for inventory_level in self.inventory_levels:
            if inventory_level.current_stock < inventory_level.reorder_point:
                send_alert(inventory_level)

# Optimize inventory ordering
def calculate_reorder_point(product, warehouse, historical_sales, lead_time):
    # Calculate the optimal reorder point based on historical sales and lead time
    safety_stock = calculate_safety_stock(historical_sales, lead_time)
    reorder_point = safety_stock + (average_daily_sales * lead_time)
    return reorder_point

def calculate_reorder_quantity(product, warehouse, historical_sales, lead_time, reorder_point):
    # Calculate the optimal reorder quantity based on historical sales, lead time, and reorder point
    economic_order_quantity = calculate_economic_order_quantity(historical_sales, holding_cost,
```

```
ordering_cost)
    reorder_quantity = max(economic_order_quantity,
product.minimum_order_quantity)
    return reorder_quantity

# Generate reports
def generate_inventory_turnover_report(inventory_levels):
    # Calculate and report on inventory turnover rates
    pass

def generate_stockout_report(inventory_levels):
    # Report on stockout occurrences and their cost implications
    pass

def generate_overstock_report(inventory_levels):
    # Report on overstock situations and their cost implications
    pass

# User interaction
class InventoryManagementUI:
    def __init__(self, inventory_tracking_app):
        self.inventory_tracking_app = inventory_tracking_app

    def display_inventory_information(self, product_id):
        # Allow users to view current stock levels, reorder
recommendations, and historical data
        pass
```

Detailed explanation of the actual code:

1. Defining the Inventory System Structure: I will create classes for Product, Warehouse, and InventoryLevel to represent the components of the inventory system. These classes will store relevant information about each entity, such as product details, warehouse locations, and

current stock levels.

2. Implementing the Inventory Tracking Application: The InventoryTrackingApp class will be responsible for initializing the inventory levels, tracking the current stock, and sending alerts when stock falls below a certain threshold. The track_inventory() method will continuously monitor the inventory levels and trigger alerts as needed.
3. Optimizing Inventory Ordering: The calculate_reorder_point() and calculate_reorder_quantity() functions will implement algorithms to determine the optimal reorder point and quantity for each product in each warehouse. These calculations will be based on historical sales data, lead times, and other relevant factors.
4. Generating Reports: The generate_inventory_turnover_report(), generate_stockout_report(), and generate_overstock_report() functions will generate the required reports on inventory performance, stockouts, and overstock situations, respectively.
5. User Interaction: The InventoryManagementUI class will provide a user-friendly interface for accessing inventory information, viewing reports, and receiving alerts. Users will be able to input product IDs or names to retrieve the desired information.

Assumptions made (:if any)

1. The company has a well-defined product catalog and warehouse locations.
2. Historical sales data and lead times are available for the inventory optimization algorithms.
3. The company has defined thresholds for reorder points and minimum order quantities.
4. The cost of holding inventory and placing orders are known.

Limitations:

1. The current implementation assumes constant lead times, which may not always be the case in real-world scenarios.
2. The demand forecasting algorithms are not included in the pseudocode, as they can be complex and require more detailed analysis of the company's sales patterns.
3. The user interface is only briefly mentioned, and a more comprehensive design would be required for a production-ready system.

Code:

class Product:

```
    def _init_(self, product_id, name, current_stock, reorder_point,
reorder_quantity):
        self.product_id = product_id
        self.name = name
        self.current_stock = current_stock
        self.reorder_point = reorder_point
        self.reorder_quantity = reorder_quantity


class Warehouse:
    def _init_(self, warehouse_id, location):
        self.warehouse_id = warehouse_id
        self.location = location
        self.products = []


def track_inventory(products):
    for product in products:
        if product.current_stock < product.reorder_point:
            print(f"Alert: {product.name} is below the reorder point. Current stock:
{product.current_stock}")
            recommend_reorder(product)


def recommend_reorder(product):
    new_stock = product.current_stock + product.reorder_quantity
    print(f"Recommended reorder for {product.name}: {product.reorder_quantity}
units. New stock level: {new_stock}")


def calculate_reorder_point(historical_sales, lead_time, desired_service_level):
    # Implement algorithms to calculate the optimal reorder point
    # based on historical sales data, lead time, and desired service level
    pass


def calculate_reorder_quantity(historical_sales, lead_time, holding_cost,
ordering_cost):
    # Implement algorithms to calculate the optimal reorder quantity
    # based on historical sales data, lead time, holding cost, and ordering cost
    pass
```

```
def generate_inventory_report(products):
    # Generate reports on inventory turnover rates, stockout occurrences, and
cost implications of overstock situations
    pass


def user_interface():
    # Define sample products and warehouses
    product1 = Product(1, "Product A", 50, 20, 30)
    product2 = Product(2, "Product B", 15, 10, 25)
    warehouse1 = Warehouse(1, "Warehouse A")
    warehouse1.products = [product1, product2]


    while True:
        user_input = input("Enter a product ID or name (or 'exit' to quit): ")
        if user_input.lower() == "exit":
            break


        # Look up the product and display current stock, reorder
recommendations, and historical data
        for product in warehouse1.products:
            if str(product.product_id) == user_input or product.name.lower() ==
user_input.lower():
                print(f"Product: {product.name}")
                print(f"Current Stock: {product.current_stock}")
                recommend_reorder(product)
                # Display historical data
                break
        else:
            print("Product not found.")


# Test the application
user_interface()
```

Sample Output / Screen Shots