package entity;

import java.util.ArrayList;

import java.util.List;

public class Customer {

private int customerID;

private String firstName;

private String lastName;

private String email;

private String phone;

private String address;

private List<Order> orders;

public Customer(int customerID, String firstName, String lastName, String email, String phone, String address) {

this.customerID = customerID;

this.firstName = firstName;

this.lastName = lastName;

this.email = email;

this.phone = phone;

this.address = address;

this.orders = new ArrayList<>();

}

// Getters and setters with validation

public int getCustomerID() { return customerID; }

public String getFirstName() { return firstName; }

public void setFirstName(String firstName) {

if (firstName == null || firstName.trim().isEmpty()) {

throw new IllegalArgumentException("First name cannot be empty");

}

this.firstName = firstName;

}

public String getLastName() { return lastName; }

public void setLastName(String lastName) {

if (lastName == null || lastName.trim().isEmpty()) {

throw new IllegalArgumentException("Last name cannot be empty");

}

this.lastName = lastName;

}

public String getEmail() { return email; }

public void setEmail(String email) {

if (email == null || !email.contains("@") || !email.contains(".")) {

throw new IllegalArgumentException("Invalid email format");

}

this.email = email;

}

public String getPhone() { return phone; }

public void setPhone(String phone) {

if (phone == null || phone.trim().isEmpty()) {

throw new IllegalArgumentException("Phone cannot be empty");

}

this.phone = phone;

}

public String getAddress() { return address; }

public void setAddress(String address) {

if (address == null || address.trim().isEmpty()) {

throw new IllegalArgumentException("Address cannot be empty");

}

this.address = address;

}

public List<Order> getOrders() { return orders; }

// Methods

public int calculateTotalOrders() {

return orders.size();

}

public void getCustomerDetails() {

System.out.println("Customer ID: " + customerID);

System.out.println("Name: " + firstName + " " + lastName);

System.out.println("Email: " + email);

System.out.println("Phone: " + phone);

System.out.println("Address: " + address);

System.out.println("Total Orders: " + calculateTotalOrders());

}

public void updateCustomerInfo(String email, String phone, String address) {

setEmail(email);

setPhone(phone);

setAddress(address);

System.out.println("Customer information updated successfully.");

}

public void addOrder(Order order) {

orders.add(order);

}

}

package entity;

public class Product {

private int productID;

private String productName;

private String description;

private double price;

public Product(int productID, String productName, String description, double price) {

this.productID = productID;

this.productName = productName;

this.description = description;

this.price = price;

}

// Getters and setters with validation

public int getProductID() { return productID; }

public String getProductName() { return productName; }

public void setProductName(String productName) {

if (productName == null || productName.trim().isEmpty()) {

throw new IllegalArgumentException("Product name cannot be empty");

}

this.productName = productName;

}

public String getDescription() { return description; }

public void setDescription(String description) {

this.description = description;

}

public double getPrice() { return price; }

public void setPrice(double price) {

if (price < 0) {

throw new IllegalArgumentException("Price cannot be negative");

}

this.price = price;

}

// Methods

public void getProductDetails() {

System.out.println("Product ID: " + productID);

System.out.println("Name: " + productName);

System.out.println("Description: " + description);

System.out.println("Price: $" + price);

}

public void updateProductInfo(String description, double price) {

setDescription(description);

setPrice(price);

System.out.println("Product information updated successfully.");

}

}

package entity;

import java.util.ArrayList;

import java.util.Date;

import java.util.List;

public class Order {

private int orderID;

private Customer customer;

private Date orderDate;

private double totalAmount;

private String status;

private List<OrderDetail> orderDetails;

public Order(int orderID, Customer customer, Date orderDate) {

this.orderID = orderID;

this.customer = customer;

this.orderDate = orderDate;

this.status = "Pending";

this.orderDetails = new ArrayList<>();

this.totalAmount = 0;

customer.addOrder(this);

}

// Getters

public int getOrderID() { return orderID; }

public Customer getCustomer() { return customer; }

public Date getOrderDate() { return orderDate; }

public double getTotalAmount() { return totalAmount; }

public String getStatus() { return status; }

public List<OrderDetail> getOrderDetails() { return orderDetails; }

// Methods

public double calculateTotalAmount() {

totalAmount = 0;

for (OrderDetail detail : orderDetails) {

totalAmount += detail.calculateSubtotal();

}

return totalAmount;

}

public void getOrderDetailsInfo() {

System.out.println("\nOrder ID: " + orderID);

System.out.println("Customer: " + customer.getFirstName() + " " + customer.getLastName());

System.out.println("Order Date: " + orderDate);

System.out.println("Status: " + status);

System.out.println("Total Amount: $" + totalAmount);

System.out.println("Products:");

for (OrderDetail detail : orderDetails) {

detail.getOrderDetailInfo();

}

}

public void updateOrderStatus(String status) {

this.status = status;

System.out.println("Order status updated to: " + status);

}

public void addOrderDetail(OrderDetail detail) {

orderDetails.add(detail);

calculateTotalAmount();

}

public void cancelOrder(Inventory inventory) {

if ("Shipped".equals(status)) {

System.out.println("Cannot cancel order that has already been shipped.");

return;

}

this.status = "Cancelled";

for (OrderDetail detail : orderDetails) {

try {

inventory.addToInventory(detail.getProduct(), detail.getQuantity());

} catch (Exception e) {

System.out.println("Error updating inventory: " + e.getMessage());

}

}

System.out.println("Order cancelled successfully. Inventory updated.");

}

}

package entity;

public class OrderDetail {

private int orderDetailID;

private Order order;

private Product product;

private int quantity;

private double discount;

public OrderDetail(int orderDetailID, Order order, Product product, int quantity) {

this.orderDetailID = orderDetailID;

this.order = order;

this.product = product;

this.quantity = quantity;

this.discount = 0;

order.addOrderDetail(this);

}

// Getters

public int getOrderDetailID() { return orderDetailID; }

public Order getOrder() { return order; }

public Product getProduct() { return product; }

public int getQuantity() { return quantity; }

public double getDiscount() { return discount; }

// Methods

public double calculateSubtotal() {

return product.getPrice() \* quantity \* (1 - discount);

}

public void getOrderDetailInfo() {

System.out.printf(" - %s x%d @ $%.2f (Subtotal: $%.2f)\n",

product.getProductName(), quantity, product.getPrice(), calculateSubtotal());

}

public void updateQuantity(int newQuantity) {

if (newQuantity <= 0) {

throw new IllegalArgumentException("Quantity must be positive");

}

this.quantity = newQuantity;

order.calculateTotalAmount();

System.out.println("Quantity updated successfully.");

}

public void addDiscount(double discountPercentage) {

if (discountPercentage < 0 || discountPercentage > 1) {

throw new IllegalArgumentException("Discount must be between 0 and 1 (0% to 100%)");

}

this.discount = discountPercentage;

order.calculateTotalAmount();

System.out.println("Discount applied successfully.");

}

}

package entity;

import java.util.ArrayList;

import java.util.Date;

import java.util.List;

public class Inventory {

private int inventoryID;

private Product product;

private int quantityInStock;

private Date lastStockUpdate;

private List<Inventory> inventoryList;

public Inventory(int inventoryID, Product product, int quantityInStock) {

this.inventoryID = inventoryID;

this.product = product;

this.quantityInStock = quantityInStock;

this.lastStockUpdate = new Date();

this.inventoryList = new ArrayList<>();

}

// Getters

public int getInventoryID() { return inventoryID; }

public Product getProduct() { return product; }

public int getQuantityInStock() { return quantityInStock; }

public Date getLastStockUpdate() { return lastStockUpdate; }

// Methods

public void addToInventory(Product product, int quantity) {

if (quantity <= 0) {

throw new IllegalArgumentException("Quantity must be positive");

}

for (Inventory item : inventoryList) {

if (item.getProduct().getProductID() == product.getProductID()) {

item.quantityInStock += quantity;

item.lastStockUpdate = new Date();

return;

}

}

inventoryList.add(new Inventory(inventoryList.size() + 1, product, quantity));

}

public void removeFromInventory(Product product, int quantity) throws InsufficientStockException {

if (quantity <= 0) {

throw new IllegalArgumentException("Quantity must be positive");

}

for (Inventory item : inventoryList) {

if (item.getProduct().getProductID() == product.getProductID()) {

if (item.quantityInStock < quantity) {

throw new InsufficientStockException("Not enough stock available");

}

item.quantityInStock -= quantity;

item.lastStockUpdate = new Date();

return;

}

}

throw new InsufficientStockException("Product not found in inventory");

}

public void updateStockQuantity(Product product, int newQuantity) {

if (newQuantity < 0) {

throw new IllegalArgumentException("Quantity cannot be negative");

}

for (Inventory item : inventoryList) {

if (item.getProduct().getProductID() == product.getProductID()) {

item.quantityInStock = newQuantity;

item.lastStockUpdate = new Date();

return;

}

}

inventoryList.add(new Inventory(inventoryList.size() + 1, product, newQuantity));

}

public boolean isProductAvailable(Product product, int quantityToCheck) {

for (Inventory item : inventoryList) {

if (item.getProduct().getProductID() == product.getProductID()) {

return item.quantityInStock >= quantityToCheck;

}

}

return false;

}

public double getInventoryValue() {

double totalValue = 0;

for (Inventory item : inventoryList) {

totalValue += item.getProduct().getPrice() \* item.quantityInStock;

}

return totalValue;

}

public List<Product> listLowStockProducts(int threshold) {

List<Product> lowStockProducts = new ArrayList<>();

for (Inventory item : inventoryList) {

if (item.quantityInStock < threshold) {

lowStockProducts.add(item.getProduct());

}

}

return lowStockProducts;

}

public List<Product> listOutOfStockProducts() {

List<Product> outOfStockProducts = new ArrayList<>();

for (Inventory item : inventoryList) {

if (item.quantityInStock == 0) {

outOfStockProducts.add(item.getProduct());

}

}

return outOfStockProducts;

}

public void listAllProducts() {

System.out.println("\nCurrent Inventory:");

for (Inventory item : inventoryList) {

System.out.printf(" - %s (ID: %d): %d in stock\n",

item.getProduct().getProductName(),

item.getProduct().getProductID(),

item.quantityInStock);

}

System.out.printf("Total Inventory Value: $%.2f\n", getInventoryValue());

}

}

package exception;

public class InvalidDataException extends Exception {

public InvalidDataException(String message) {

super(message);

}

}

package exception;

public class InsufficientStockException extends Exception {

public InsufficientStockException(String message) {

super(message);

}

}

package exception;

public class IncompleteOrderException extends Exception {

public IncompleteOrderException(String message) {

super(message);

}

}

package exception;

public class PaymentFailedException extends Exception {

public PaymentFailedException(String message) {

super(message);

}

}

package exception;

public class AuthenticationException extends Exception {

public AuthenticationException(String message) {

super(message);

}

}

package exception;

public class AuthorizationException extends Exception {

public AuthorizationException(String message) {

super(message);

}

}

package dao;

import entity.\*;

import exception.\*;

import java.util.List;

public interface TechShopService {

// Customer operations

void addCustomer(Customer customer) throws InvalidDataException;

Customer getCustomerById(int customerId);

void updateCustomer(Customer customer) throws InvalidDataException;

List<Customer> getAllCustomers();

// Product operations

void addProduct(Product product) throws InvalidDataException;

Product getProductById(int productId);

void updateProduct(Product product) throws InvalidDataException;

List<Product> getAllProducts();

// Order operations

void placeOrder(Order order) throws IncompleteOrderException, InsufficientStockException;

Order getOrderById(int orderId);

void updateOrderStatus(int orderId, String status);

void cancelOrder(int orderId) throws Exception;

List<Order> getOrdersByCustomer(int customerId);

// Inventory operations

void addToInventory(Product product, int quantity) throws InvalidDataException;

void removeFromInventory(Product product, int quantity) throws InsufficientStockException;

int getProductStock(int productId);

List<Product> getLowStockProducts(int threshold);

List<Product> getOutOfStockProducts();

}

package dao;

import entity.\*;

import exception.\*;

import java.util.ArrayList;

import java.util.Date;

import java.util.List;

public class TechShopServiceImpl implements TechShopService {

private List<Customer> customers = new ArrayList<>();

private List<Product> products = new ArrayList<>();

private List<Order> orders = new ArrayList<>();

private Inventory inventory = new Inventory(1, null, 0);

@Override

public void addCustomer(Customer customer) throws InvalidDataException {

try {

// Check for duplicate email

for (Customer c : customers) {

if (c.getEmail().equalsIgnoreCase(customer.getEmail())) {

throw new InvalidDataException("Email already exists");

}

}

customers.add(customer);

} catch (Exception e) {

throw new InvalidDataException("Invalid customer data: " + e.getMessage());

}

}

@Override

public Customer getCustomerById(int customerId) {

for (Customer customer : customers) {

if (customer.getCustomerID() == customerId) {

return customer;

}

}

return null;

}

@Override

public void updateCustomer(Customer customer) throws InvalidDataException {

try {

Customer existing = getCustomerById(customer.getCustomerID());

if (existing != null) {

existing.setFirstName(customer.getFirstName());

existing.setLastName(customer.getLastName());

existing.setEmail(customer.getEmail());

existing.setPhone(customer.getPhone());

existing.setAddress(customer.getAddress());

} else {

throw new InvalidDataException("Customer not found");

}

} catch (Exception e) {

throw new InvalidDataException("Error updating customer: " + e.getMessage());

}

}

@Override

public List<Customer> getAllCustomers() {

return new ArrayList<>(customers);

}

@Override

public void addProduct(Product product) throws InvalidDataException {

try {

// Check for duplicate product name

for (Product p : products) {

if (p.getProductName().equalsIgnoreCase(product.getProductName())) {

throw new InvalidDataException("Product name already exists");

}

}

products.add(product);

inventory.addToInventory(product, 0); // Add to inventory with 0 quantity

} catch (Exception e) {

throw new InvalidDataException("Invalid product data: " + e.getMessage());

}

}

@Override

public Product getProductById(int productId) {

for (Product product : products) {

if (product.getProductID() == productId) {

return product;

}

}

return null;

}

@Override

public void updateProduct(Product product) throws InvalidDataException {

try {

Product existing = getProductById(product.getProductID());

if (existing != null) {

existing.setProductName(product.getProductName());

existing.setDescription(product.getDescription());

existing.setPrice(product.getPrice());

} else {

throw new InvalidDataException("Product not found");

}

} catch (Exception e) {

throw new InvalidDataException("Error updating product: " + e.getMessage());

}

}

@Override

public List<Product> getAllProducts() {

return new ArrayList<>(products);

}

@Override

public void placeOrder(Order order) throws IncompleteOrderException, InsufficientStockException {

if (order.getOrderDetails().isEmpty()) {

throw new IncompleteOrderException("Order must have at least one item");

}

// Check inventory and reserve items

for (OrderDetail detail : order.getOrderDetails()) {

if (!inventory.isProductAvailable(detail.getProduct(), detail.getQuantity())) {

throw new InsufficientStockException("Product " + detail.getProduct().getProductName() +

" has insufficient stock");

}

}

// Process order

for (OrderDetail detail : order.getOrderDetails()) {

inventory.removeFromInventory(detail.getProduct(), detail.getQuantity());

}

orders.add(order);

order.updateOrderStatus("Processing");

}

@Override

public Order getOrderById(int orderId) {

for (Order order : orders) {

if (order.getOrderID() == orderId) {

return order;

}

}

return null;

}

@Override

public void updateOrderStatus(int orderId, String status) {

Order order = getOrderById(orderId);

if (order != null) {

order.updateOrderStatus(status);

}

}

@Override

public void cancelOrder(int orderId) throws Exception {

Order order = getOrderById(orderId);

if (order != null) {

order.cancelOrder(inventory);

orders.remove(order);

} else {

throw new Exception("Order not found");

}

}

@Override

public List<Order> getOrdersByCustomer(int customerId) {

List<Order> customerOrders = new ArrayList<>();

for (Order order : orders) {

if (order.getCustomer().getCustomerID() == customerId) {

customerOrders.add(order);

}

}

return customerOrders;

}

@Override

public void addToInventory(Product product, int quantity) throws InvalidDataException {

try {

inventory.addToInventory(product, quantity);

} catch (Exception e) {

throw new InvalidDataException("Error adding to inventory: " + e.getMessage());

}

}

@Override

public void removeFromInventory(Product product, int quantity) throws InsufficientStockException {

inventory.removeFromInventory(product, quantity);

}

@Override

public int getProductStock(int productId) {

Product product = getProductById(productId);

if (product != null) {

for (Inventory item : inventory.getInventoryList()) {

if (item.getProduct().getProductID() == productId) {

return item.getQuantityInStock();

}

}

}

return 0;

}

@Override

public List<Product> getLowStockProducts(int threshold) {

return inventory.listLowStockProducts(threshold);

}

@Override

public List<Product> getOutOfStockProducts() {

return inventory.listOutOfStockProducts();

}

}

package util;

import java.io.FileInputStream;

import java.io.IOException;

import java.util.Properties;

public class DBPropertyUtil {

public static String getConnectionString(String propertyFileName) {

Properties properties = new Properties();

try (FileInputStream input = new FileInputStream(propertyFileName)) {

properties.load(input);

return String.format("jdbc:mysql://%s:%s/%s?user=%s&password=%s",

properties.getProperty("db.host"),

properties.getProperty("db.port"),

properties.getProperty("db.name"),

properties.getProperty("db.user"),

properties.getProperty("db.password"));

} catch (IOException e) {

System.err.println("Error reading property file: " + e.getMessage());

return null;

}

}

}

package util;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

public class DBConnUtil {

public static Connection getConnection(String connectionString) {

try {

return DriverManager.getConnection(connectionString);

} catch (SQLException e) {

System.err.println("Error establishing database connection: " + e.getMessage());

return null;

}

}

}

package main;

import dao.TechShopService;

import dao.TechShopServiceImpl;

import entity.\*;

import exception.\*;

import java.util.Date;

import java.util.List;

public class MainModule {

public static void main(String[] args) {

TechShopService techShop = new TechShopServiceImpl();

try {

// Create sample products

Product laptop = new Product(1, "Laptop", "High-performance laptop", 999.99);

Product phone = new Product(2, "Smartphone", "Latest smartphone model", 699.99);

Product headphones = new Product(3, "Headphones", "Noise-cancelling headphones", 199.99);

techShop.addProduct(laptop);

techShop.addProduct(phone);

techShop.addProduct(headphones);

// Add inventory

techShop.addToInventory(laptop, 10);

techShop.addToInventory(phone, 20);

techShop.addToInventory(headphones, 15);

// Create a customer

Customer customer = new Customer(1, "John", "Doe", "john.doe@example.com", "1234567890", "123 Main St");

techShop.addCustomer(customer);

// Display inventory

System.out.println("=== Initial Inventory ===");

techShop.getOutOfStockProducts().forEach(p -> System.out.println(p.getProductName() + " is out of stock"));

System.out.println("Low stock products:");

techShop.getLowStockProducts(5).forEach(p -> System.out.println(p.getProductName()));

// Create an order

Order order = new Order(1, customer, new Date());

order.addOrderDetail(new OrderDetail(1, order, laptop, 1));

order.addOrderDetail(new OrderDetail(2, order, headphones, 2));

// Place the order

techShop.placeOrder(order);

System.out.println("\n=== Order Placed ===");

order.getOrderDetailsInfo();

// Display updated inventory

System.out.println("\n=== Updated Inventory ===");

System.out.println("Laptop stock: " + techShop.getProductStock(1));

System.out.println("Headphones stock: " + techShop.getProductStock(3));

// Update order status

techShop.updateOrderStatus(1, "Shipped");

System.out.println("\n=== Updated Order Status ===");

techShop.getOrderById(1).getOrderDetailsInfo();

// Try to cancel shipped order (should fail)

try {

techShop.cancelOrder(1);

} catch (Exception e) {

System.out.println("\nError cancelling order: " + e.getMessage());

}

// Create another order

Order order2 = new Order(2, customer, new Date());

order2.addOrderDetail(new OrderDetail(3, order2, phone, 3));

order2.addOrderDetail(new OrderDetail(4, order2, headphones, 1));

// Place the second order

techShop.placeOrder(order2);

System.out.println("\n=== Second Order Placed ===");

order2.getOrderDetailsInfo();

// Cancel the second order (should succeed)

techShop.cancelOrder(2);

System.out.println("\n=== After Cancelling Second Order ===");

System.out.println("Headphones stock: " + techShop.getProductStock(3));

System.out.println("Phone stock: " + techShop.getProductStock(2));

// Get all customer orders

System.out.println("\n=== Customer Orders ===");

List<Order> customerOrders = techShop.getOrdersByCustomer(1);

customerOrders.forEach(Order::getOrderDetailsInfo);

// Test exception handling

System.out.println("\n=== Testing Exception Handling ===");

try {

techShop.addCustomer(new Customer(2, "Jane", "Doe", "invalid-email", "9876543210", "456 Oak St"));

} catch (InvalidDataException e) {

System.out.println("Error adding customer: " + e.getMessage());

}

try {

Order invalidOrder = new Order(3, customer, new Date());

techShop.placeOrder(invalidOrder);

} catch (IncompleteOrderException e) {

System.out.println("Error placing order: " + e.getMessage());

}

try {

Order overstockOrder = new Order(4, customer, new Date());

overstockOrder.addOrderDetail(new OrderDetail(5, overstockOrder, laptop, 100));

techShop.placeOrder(overstockOrder);

} catch (InsufficientStockException e) {

System.out.println("Error placing order: " + e.getMessage());

}

} catch (Exception e) {

System.err.println("An error occurred: " + e.getMessage());

e.printStackTrace();

}

}

}

Database.properties

db.hostname=localhost

db.port=3306

db.name=ecommerce\_db

db.username=root

db.password=Nijandhan