Міністерство освіти і науки України

Національний університет «Львівська політехніка»

Інститут комп’ютерних наук та інформаційних технологій

Кафедра автоматизованих систем управління



**Звіт**

до лабораторної роботи № 7 з

курсу

*Прикладне програмувння*  на

тему:

**« Комплексна лабораторна робота »**

Виконав: студент ОІ-24

**Стащишин Юрій**

Прийняв:

**Чорненький В. Я.**

Львів – 2024

***Лабораторна робота № 7***

**Тема:** Комплексна лабораторна робота

**Мета роботи:** Покрити програму юніт тестами

**Варіант - 13**

Покрийте вашу програму юніт тестами (coverage >=90%).

**Код програми**

package truck.command;  
  
import truck.model.Coffee;  
import truck.model.Shop;  
  
public class AddCoffeeCommand implements Command {  
 private Shop shop;  
 private Coffee coffee;  
  
 public AddCoffeeCommand(Shop shop) {  
 this.shop = shop;  
 }  
  
 // Метод для встановлення конкретної кави, яку потрібно додати  
 public void setCoffee(Coffee coffee) {  
 this.coffee = coffee;  
 }  
  
 @Override  
 public void execute() {  
 if (coffee != null) {  
 shop.addCoffee(coffee);  
 } else {  
 System.*out*.println("Необхідно встановити каву перед додаванням.");  
 }  
 }  
}

package truck.command;  
  
public interface Command {  
 void execute();  
}

package truck.command;  
  
public class ExitCommand implements Command{  
 @Override  
 public void execute(){  
 System.*out*.println("Вихід з програми...");  
 System.*exit*(0);  
 }  
}

package truck.command;  
  
import truck.model.Coffee;  
import truck.model.Truck;  
  
public class LoadTruckCommand implements Command{  
 private Truck truck;  
 private Coffee coffee;  
 private int quantity;  
  
 public LoadTruckCommand(Truck truckManager) {  
 this.truck = truckManager;  
 }  
  
 // Метод для встановлення кави, яку необхідно завантажити  
 public void setCoffee(Coffee coffee, int quantity) {  
 this.coffee = coffee;  
 this.quantity = quantity;  
 }  
  
 @Override  
 public void execute(){  
 if (coffee != null && quantity > 0) {  
 truck.loadTruck(coffee, quantity);  
 } else {  
 System.*out*.println("Необхідно встановити каву та кількість перед завантаженням.");  
 }  
 }  
}

package truck.command;  
  
import truck.model.Truck;  
  
public class SearchCoffeeCommand implements Command{  
 Truck truck;  
  
 public SearchCoffeeCommand(Truck truck) {  
 this.truck = truck;  
 }  
  
 @Override  
 public void execute(){  
 truck.searchCoffee();  
 }  
}

package truck.command;  
  
import truck.model.Shop;  
  
public class ShowAvailableCoffeeCommand implements Command{  
 Shop shop;  
  
 public ShowAvailableCoffeeCommand(Shop shop) {  
 this.shop = shop;  
 }  
  
 @Override  
 public void execute(){  
 shop.showAvailableCoffee();  
 }  
}

package truck.command;  
  
import truck.model.Truck;  
  
public class SortCoffeeCommand implements Command{  
 Truck truck;  
  
 public SortCoffeeCommand(Truck truck) {  
 this.truck = truck;  
 }  
  
 @Override  
 public void execute(){  
 truck.sortCoffee();  
 }  
}

package truck.model;  
  
import truck.command.Command;  
  
// Invoker  
public class Carrier {  
 Command load;  
 Command search;  
 Command sort;  
  
 public Carrier(Command load, Command search, Command sort) {  
 this.load = load;  
 this.search = search;  
 this.sort = sort;  
 }  
  
 public void loadTruck(){  
 load.execute();  
 }  
  
 public void searchCoffee(){  
 search.execute();  
 }  
  
 public void sortCoffee(){  
 sort.execute();  
 }  
}

package truck.model;  
  
public class Coffee {  
 private String name;  
 private double weight;  
 private double price;  
 private double volume;  
 private Package packaging;  
 private int quantity; // Кількість упаковок  
 private int quantityInTruck = 0;  
  
 public Coffee(String name, double weight, double price, double volume, Package packaging, int quantity) {  
 this.name = name;  
 this.weight = weight;  
 this.price = price;  
 this.volume = volume;  
 this.packaging = packaging;  
 this.quantity = quantity;  
 }  
  
  
 public String getName() {  
 return name;  
 }  
  
 public double getWeight() {  
 return weight;  
 }  
  
 public double getPrice() {  
 return price;  
 }  
  
 public int getQuantity() {  
 return quantity;  
 }  
  
 public void setQuantity(int quantity) {  
 this.quantity = quantity;  
 }  
  
 public int getQuantityInTruck() {  
 return quantityInTruck;  
 }  
  
 public void setQuantityInTruck(int quantityInTruck) {  
 this.quantityInTruck = quantityInTruck;  
 }  
  
 public double getTotalVolume() {  
 return packaging.getVolume();  
 }  
  
 @Override  
 public String toString() {  
 return String.*format*("%s: вага = %.2f г, ціна = %.2f грн, об'єм = %.2f л, кількість упаковок = %d",  
 name, weight, price, packaging.getVolume(), quantity);  
 }  
  
 public String toStringInTruck() {  
 return String.*format*("%s: вага = %.2f г, ціна = %.2f грн, об'єм = %.2f л, кількість упаковок = %d",  
 name, weight, price, packaging.getVolume(), quantityInTruck);  
 }  
  
}

package truck.model;  
  
public class GrainCoffee extends Coffee{  
 public GrainCoffee(String name, double weight, double price, double volume, Package packaging, int quantity) {  
 super(name, weight, price, volume, packaging, quantity);  
 }  
}

package truck.model;  
  
public class GroundCoffee extends Coffee{  
 public GroundCoffee(String name, double weight, double price, double volume, Package packaging, int quantity) {  
 super(name, weight, price, volume, packaging, quantity);  
 }  
}

package truck.model;  
  
public class InstantCoffee extends Coffee{  
 public InstantCoffee(String name, double weight, double price, double volume, Package packaging, int quantity) {  
 super(name, weight, price, volume, packaging, quantity);  
 }  
}

package truck.model;  
  
// Клас для збереження даних про упаковку  
public class Package {  
 private String material;  
 private double weight;  
 private double volume;  
  
 public Package(String material, double weight, double volume) {  
 this.material = material;  
 this.weight = weight;  
 this.volume = volume;  
 }  
  
 public String getMaterial() {  
 return material;  
 }  
  
 public double getWeight() {  
 return weight;  
 }  
  
 public double getVolume() {  
 return volume;  
 }  
  
 @Override  
 public String toString() {  
 return String.*format*("Упаковка: матеріал = %s, вага = %.2f г, об'єм = %.2f л", material, weight, volume);  
 }  
}

package truck.model;  
  
import truck.command.Command;  
  
public class Seller {  
 Command add;  
 Command show;  
  
 public Seller(Command show, Command add) {  
 this.add = add;  
 this.show = show;  
 }  
  
 public void addCoffee(){  
 add.execute();  
 }  
  
 public void showCoffee(){  
 show.execute();  
 }  
}

package truck.model;  
  
import java.io.BufferedReader;  
import java.io.FileReader;  
import java.io.IOException;  
import java.util.ArrayList;  
import java.util.List;  
  
public class Shop {  
 private List<Coffee> availableCoffee;  
  
 public Shop() {  
 this.availableCoffee = new ArrayList<>();  
 initializeDefaultCoffees();  
 }  
  
 private void initializeDefaultCoffees() {  
 // Створення кількох видів кави за замовчуванням  
 Package package1 = new Package("Банка", 0.5, 1.0);  
 Package package2 = new Package("Пакет", 0.2, 0.5);  
  
 availableCoffee.add(new GrainCoffee("Зернова кава", 500, 100, 1, package1, 1000));  
 availableCoffee.add(new GroundCoffee("Мелена кава", 260, 50, 0.5, package2, 500));  
 availableCoffee.add(new InstantCoffee("Розчинна кава", 100, 30, 0.25, package2, 2000));  
 }  
  
 // Завантажує каву з файлу та додає її до списку доступних кав  
 public void loadCoffeesFromFile(String filename) {  
 try (BufferedReader br = new BufferedReader(new FileReader(filename))) {  
 String line;  
 while ((line = br.readLine()) != null) {  
 String[] parts = line.split(",");  
 if (parts.length == 8) {  
 String name = parts[0];  
 double weight = Double.*parseDouble*(parts[1]);  
 double price = Double.*parseDouble*(parts[2]);  
 double volume = Double.*parseDouble*(parts[3]);  
 String packagingMaterial = parts[4];  
 double packageWeight = Double.*parseDouble*(parts[5]);  
 double packageVolume = Double.*parseDouble*(parts[6]);  
 int quantity = Integer.*parseInt*(parts[7]);  
  
 Package packaging = new Package(packagingMaterial, packageWeight, packageVolume);  
 Coffee coffee = new GrainCoffee(name, weight, price, volume, packaging, quantity);  
 availableCoffee.add(coffee);  
 }  
 }  
 System.*out*.println("Каву з файлу добавлено");  
 } catch (IOException e) {  
 System.*out*.println("Помилка при зчитуванні файлу: " + e.getMessage());  
 }  
 }  
  
 // Додає новий об'єкт кави до списку доступних кав  
 public void addCoffee(Coffee coffee){  
 availableCoffee.add(coffee);  
 System.*out*.println("Додана кава: " + coffee);  
 }  
  
 // Виводить список доступної кави  
 public void showAvailableCoffee(){  
 if (availableCoffee.isEmpty()) {  
 System.*out*.println("Кава відсутня.");  
 } else {  
 System.*out*.println("Кава на вибір:");  
 for (Coffee coffee : availableCoffee) {  
 System.*out*.println(coffee);  
 }  
 }  
 }  
  
 public void setAvailableCoffee(List<Coffee> availableCoffee) {  
 this.availableCoffee = availableCoffee;  
 }  
  
 public List<Coffee> getAvailableCoffee() {  
 return availableCoffee;  
 }  
  
  
}

package truck.model;  
  
import java.util.\*;  
  
// Reciver  
public class Truck {  
 private List<Coffee> loadedCoffees;  
 private double maxVolume; // в літрах (1м^3 = 1000л)  
 private double currentVolume;  
  
 public Truck(double maxVolume) {  
 this.loadedCoffees = new ArrayList<>();  
 this.maxVolume = maxVolume;  
 this.currentVolume = 0;  
 }  
  
 // Метод для завантаження фургону кавою  
 public void loadTruck(Coffee coffee, int quantity){  
 double coffeeVolume = coffee.getTotalVolume() \* quantity;  
  
 if (currentVolume + coffeeVolume <= maxVolume) {  
 int currentQuantity = coffee.getQuantity() - quantity;  
 coffee.setQuantityInTruck(quantity + coffee.getQuantityInTruck());  
 loadedCoffees.add(coffee);  
 currentVolume += coffeeVolume;  
 System.out.println(quantity + " упаковок " + coffee.getName() + " завантажено у фургон. Поточний об'єм: " + currentVolume + "/" + maxVolume);  
 coffee.setQuantity(currentQuantity);  
 } else {  
 System.out.println("Недостатньо місця у фургоні для " + quantity + " упаковок " + coffee.getName() + ". Поточний об'єм: " + currentVolume + "/" + maxVolume);  
 }  
 }  
  
 // Метод для сортування кави на основі співвідношення ціни та ваги  
 public void sortCoffee(){  
 if (loadedCoffees.isEmpty()) {  
 System.out.println("Фургон порожній. Немає кави для сортування.");  
 return;  
 }  
  
 Collections.sort(loadedCoffees, new Comparator<Coffee>() {  
 @Override  
 public int compare(Coffee c1, Coffee c2) {  
 double ratio1 = c1.getPrice() / c1.getWeight();  
 double ratio2 = c2.getPrice() / c2.getWeight();  
 return Double.compare(ratio1, ratio2);  
 }  
 });  
  
  
 System.out.println("Кава відсортована за співвідношенням ціни до ваги:");  
 for (Coffee coffee : loadedCoffees) {  
 System.out.println(coffee.toStringInTruck());  
 }  
 }  
  
 // Метод для знаходження товару у фургоні  
 public void searchCoffee(){  
 if (loadedCoffees.isEmpty()) {  
 System.out.println("Фургон порожній. Немає кави для пошуку.");  
 return;  
 }  
  
 Scanner scanner = new Scanner(System.in);  
 System.out.print("Введіть назву кави для пошуку: ");  
 String coffeeName = scanner.nextLine();  
  
 boolean found = false;  
 for (Coffee coffee : loadedCoffees) {  
 if (coffee.getName().equalsIgnoreCase(coffeeName)) {  
 System.out.println("Кава знайдена: " + coffee);  
 found = true;  
 break;  
 }  
 }  
  
 if (!found) {  
 System.out.println("Кава з назвою \"" + coffeeName + "\" не знайдена у фургоні.");  
 }  
 }  
  
 public double getCurrentVolume() {  
 return currentVolume;  
 }  
  
 public List<Coffee> getLoadedCoffees() {  
 return loadedCoffees;  
 }  
  
 public double getMaxVolume() {  
 return maxVolume;  
 }  
  
  
}

package truck;  
  
import truck.command.\*;  
import truck.model.\*;  
import truck.model.Package;  
  
import java.util.InputMismatchException;  
import java.util.List;  
import java.util.Scanner;  
  
public class ConsoleMenu{  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
 Truck truck = new Truck(2000);  
 Shop shop = new Shop();  
  
 LoadTruckCommand loadCommand = new LoadTruckCommand(truck);  
 Command searchCommand = new SearchCoffeeCommand(truck);  
 Command sortCommand = new SortCoffeeCommand(truck);  
 AddCoffeeCommand addCoffeeCommand = new AddCoffeeCommand(shop);  
 Command showCoffeeCommand = new ShowAvailableCoffeeCommand(shop);  
 Command exitCommand = new ExitCommand();  
  
 Carrier carrier = new Carrier(loadCommand, searchCommand, sortCommand);  
 Seller seller = new Seller(showCoffeeCommand, addCoffeeCommand);  
  
 boolean running = true;  
 while (running) {  
 System.*out*.println();  
 System.*out*.println("Меню:");  
 System.*out*.println("1. Завантажити фургон кавою");  
 System.*out*.println("2. Показати відсортовану каву в фургоні");  
 System.*out*.println("3. Шукати каву");  
 System.*out*.println("4. Додати каву в магазин");  
 System.*out*.println("5. Показати доступну каву в магазині");  
 System.*out*.println("6. Завнтажити дані про каву з файлу");  
 System.*out*.println("7. Вихід");  
  
 System.*out*.print("Оберіть опцію: ");  
  
 int choice = -1;  
 while (true) {  
 try {  
 choice = scanner.nextInt();  
 if (choice < 1 || choice > 7) {  
 System.*out*.println("Будь ласка, виберіть число від 1 до 7.");  
 } else {  
 break;  
 }  
 } catch (InputMismatchException e) {  
 System.*out*.println("Помилка: Введіть коректне число.");  
 scanner.next();  
 }  
 }  
  
 switch (choice) {  
 case 1:  
 Coffee load = *selectCoffee*(scanner, shop);  
 if (load != null) {  
 System.*out*.print("Введіть суму на яку хочете зробити замовлення: ");  
 int sum = scanner.nextInt();  
 int quantity = (int) (sum / load.getPrice());  
 if (quantity > load.getQuantity()){  
 System.*out*.println("Недостатньо кави в магазині");  
 break;  
 }  
 loadCommand.setCoffee(load, quantity);  
 loadCommand.execute();  
 } else {  
 System.*out*.println("Помилка: Кава не вибрана.");  
 }  
 break;  
 case 2:  
 carrier.sortCoffee();  
 break;  
 case 3:  
 carrier.searchCoffee();  
 break;  
 case 4:  
 Coffee coffee = *createCoffee*(scanner);  
 if (coffee != null) {  
 addCoffeeCommand.setCoffee(coffee);  
 seller.addCoffee();  
 }  
 break;  
 case 5:  
 seller.showCoffee();  
 break;  
 case 6:  
 shop.loadCoffeesFromFile("NotCoffees.txt");  
 break;  
 case 7:  
 exitCommand.execute();  
 break;  
 default:  
 System.*out*.println("Невірний вибір, спробуйте ще раз.");  
 }  
 }  
  
 scanner.close();  
 }  
  
 private static Coffee selectCoffee(Scanner scanner, Shop shop){  
 // Виведення списку доступної кави  
 System.*out*.println("Доступна кава в магазині:");  
 List<Coffee> availableCoffee = shop.getAvailableCoffee();  
  
 if (availableCoffee.isEmpty()) {  
 System.*out*.println("Кава відсутня.");  
 return null;  
 }  
  
 // Відображення кави з індексами для вибору  
 for (int i = 0; i < availableCoffee.size(); i++) {  
 System.*out*.printf("%d. %s%n", i + 1, availableCoffee.get(i));  
 }  
  
 int choice = -1;  
 while (true) {  
 System.*out*.print("Оберіть номер кави для завантаження у фургон: ");  
 try {  
 choice = scanner.nextInt();  
 if (choice > 0 && choice <= availableCoffee.size()) {  
 return availableCoffee.get(choice - 1);  
 } else {  
 System.*out*.println("Невірний вибір. Будь ласка, виберіть номер від 1 до " + availableCoffee.size() + ".");  
 }  
 } catch (InputMismatchException e) {  
 System.*out*.println("Помилка: Введіть коректне число.");  
 scanner.next();  
 }  
 }  
 }  
  
 private static Coffee createCoffee(Scanner scanner) {  
 System.*out*.println("Виберіть тип кави: ");  
 System.*out*.println("1. Grain Coffee");  
 System.*out*.println("2. Ground Coffee");  
 System.*out*.println("3. Instant Coffee");  
  
 int typeChoice = -1;  
 while (true) {  
 try {  
 typeChoice = scanner.nextInt();  
 if (typeChoice < 1 || typeChoice > 3) {  
 System.*out*.println("Невірний вибір. Будь ласка, виберіть тип кави від 1 до 3.");  
 } else {  
 break; // Вихід з циклу, якщо введено коректне число  
 }  
 } catch (InputMismatchException e) {  
 System.*out*.println("Помилка: Введіть коректне число.");  
 scanner.next(); // Очищення буфера  
 }  
 }  
  
 scanner.nextLine(); // Очищення буфера після введення числа  
  
 System.*out*.print("Введіть назву кави: ");  
 String name = scanner.nextLine();  
  
 double weight = -1;  
 while (weight < 0) {  
 System.*out*.print("Введіть вагу кави (в грамах): ");  
 try {  
 weight = scanner.nextDouble();  
 if (weight < 0) {  
 System.*out*.println("Вага не може бути негативною. Спробуйте ще раз.");  
 }  
 } catch (InputMismatchException e) {  
 System.*out*.println("Помилка: Введіть коректну вагу.");  
 scanner.next(); // Очищення буфера  
 }  
 }  
  
 double price = -1;  
 while (price < 0) {  
 System.*out*.print("Введіть ціну кави (в грн): ");  
 try {  
 price = scanner.nextDouble();  
 if (price < 0) {  
 System.*out*.println("Ціна не може бути негативною. Спробуйте ще раз.");  
 }  
 } catch (InputMismatchException e) {  
 System.*out*.println("Помилка: Введіть коректну ціну.");  
 scanner.next(); // Очищення буфера  
 }  
 }  
  
 double volume = -1;  
 while (volume < 0) {  
 System.*out*.print("Введіть об'єм кави (в літрах): ");  
 try {  
 volume = scanner.nextDouble();  
 if (volume < 0) {  
 System.*out*.println("Об'єм не може бути негативним. Спробуйте ще раз.");  
 }  
 } catch (InputMismatchException e) {  
 System.*out*.println("Помилка: Введіть коректний об'єм.");  
 scanner.next(); // Очищення буфера  
 }  
 }  
  
 // Вибір типу упаковки  
 System.*out*.println("Виберіть тип упаковки: ");  
 System.*out*.println("1. Банка");  
 System.*out*.println("2. Пакет");  
  
 int packagingChoice = -1;  
 while (true) {  
 try {  
 packagingChoice = scanner.nextInt();  
 if (packagingChoice < 1 || packagingChoice > 2) {  
 System.*out*.println("Невірний вибір типу упаковки. Будь ласка, виберіть 1 або 2.");  
 } else {  
 break; // Вихід з циклу, якщо введено коректне число  
 }  
 } catch (InputMismatchException e) {  
 System.*out*.println("Помилка: Введіть коректне число.");  
 scanner.next(); // Очищення буфера  
 }  
 }  
  
 String packagingMaterial = (packagingChoice == 1) ? "Банка" : "Пакет";  
  
 double packageWeight = -1;  
 while (packageWeight < 0) {  
 System.*out*.print("Введіть вагу упаковки (в грамах): ");  
 try {  
 packageWeight = scanner.nextDouble();  
 if (packageWeight < 0) {  
 System.*out*.println("Вага упаковки не може бути негативною. Спробуйте ще раз.");  
 }  
 } catch (InputMismatchException e) {  
 System.*out*.println("Помилка: Введіть коректну вагу упаковки.");  
 scanner.next(); // Очищення буфера  
 }  
 }  
  
 double packageVolume = -1;  
 while (packageVolume < 0) {  
 System.*out*.print("Введіть об'єм упаковки (в літрах): ");  
 try {  
 packageVolume = scanner.nextDouble();  
 if (packageVolume < 0) {  
 System.*out*.println("Об'єм упаковки не може бути негативним. Спробуйте ще раз.");  
 }  
 } catch (InputMismatchException e) {  
 System.*out*.println("Помилка: Введіть коректний об'єм упаковки.");  
 scanner.next(); // Очищення буфера  
 }  
 }  
  
 int quantity = -1;  
 while (quantity < 0) {  
 System.*out*.print("Введіть кількість упаковок: ");  
 try {  
 quantity = scanner.nextInt();  
 if (quantity < 0) {  
 System.*out*.println("Кількість не може бути негативною. Спробуйте ще раз.");  
 }  
 } catch (InputMismatchException e) {  
 System.*out*.println("Помилка: Введіть коректну кількість упаковок.");  
 scanner.next(); // Очищення буфера  
 }  
 }  
  
 Package packaging = new Package(packagingMaterial, packageWeight, packageVolume);  
  
 switch (typeChoice) {  
 case 1:  
 return new GrainCoffee(name, weight, price, volume, packaging, quantity);  
 case 2:  
 return new GroundCoffee(name, weight, price, volume, packaging, quantity);  
 case 3:  
 return new InstantCoffee(name, weight, price, volume, packaging, quantity);  
 default:  
 System.*out*.println("Невірний вибір типу кави.");  
 return null;  
 }  
 }  
}

**Тести**

package truck.command;  
  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.\*;  
import java.io.ByteArrayOutputStream;  
import java.io.PrintStream;  
import truck.model.Coffee;  
import truck.model.Package;  
import truck.model.Shop;  
import truck.command.AddCoffeeCommand;  
  
class AddCoffeeCommandTest {  
  
 private Shop shop;  
 private AddCoffeeCommand addCoffeeCommand;  
  
 @BeforeEach  
 void setUp() {  
 // Ініціалізація об'єктів перед кожним тестом  
 shop = new Shop();  
 addCoffeeCommand = new AddCoffeeCommand(shop);  
 }  
  
 @Test  
 void testExecute\_withValidCoffee() {  
 // Створення об'єкта кави  
 truck.model.Package package1 = new Package("Банка", 0.5, 1.0);  
 Coffee coffee = new Coffee("Arabica", 500, 100, 1, package1, 1); // Назва, вага, ціна  
 addCoffeeCommand.setCoffee(coffee);  
  
 // Виконання команди додавання кави  
 addCoffeeCommand.execute();  
  
 // Перевірка, чи каву було додано до магазину  
 *assertTrue*(shop.getAvailableCoffee().contains(coffee), "Кава повинна бути додана до магазину");  
 }  
  
 @Test  
 void testExecute\_withoutCoffee() {  
 // Підготовка для тесту, коли кава не була встановлена  
 ByteArrayOutputStream outputStream = new ByteArrayOutputStream();  
 PrintStream originalOut = System.*out*; // Зберігаємо оригінальний потік виведення  
 System.*setOut*(new PrintStream(outputStream)); // Перенаправляємо потік виведення  
  
 // Виконання команди додавання кави без її встановлення  
 addCoffeeCommand.execute();  
  
 // Перевірка, що виведено правильне повідомлення  
 String output = outputStream.toString().trim();  
 *assertEquals*("Необхідно встановити каву перед додаванням.", output);  
  
 // Відновлення оригінального потоку виведення  
 System.*setOut*(originalOut);  
 }  
}

package truck.command;  
  
import static org.junit.jupiter.api.Assertions.\*;  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.\*;  
import java.io.ByteArrayOutputStream;  
import java.io.PrintStream;  
import truck.model.Coffee;  
import truck.model.Package;  
import truck.model.Truck;  
import truck.command.LoadTruckCommand;  
  
class LoadTruckCommandTest {  
  
 private Truck truck;  
 private LoadTruckCommand loadTruckCommand;  
  
 @BeforeEach  
 void setUp() {  
 // Ініціалізація об'єктів перед кожним тестом  
 truck = new Truck(10000);  
 loadTruckCommand = new LoadTruckCommand(truck);  
 }  
  
 @Test  
 void testExecute\_withValidCoffeeAndQuantity() {  
 // Створення об'єкта кави  
 truck.model.Package package1 = new Package("Банка", 0.5, 1.0);  
 Coffee coffee = new Coffee("Robusta", 500, 100, 1, package1, 20);  
 int quantity = 10;  
 loadTruckCommand.setCoffee(coffee, quantity);  
  
 // Виконання команди завантаження кави в фургон  
 loadTruckCommand.execute();  
  
 // Перевірка, чи кава була додана в фургон  
 *assertTrue*(truck.getLoadedCoffees().contains(coffee), "Кава повинна бути завантажена в фургон");  
 *assertEquals*(quantity, coffee.getQuantity(), "Кількість кави повинна бути правильна");  
 }  
  
 @Test  
 void testExecute\_withoutCoffee() {  
 // Підготовка для тесту, коли кава не була встановлена  
 ByteArrayOutputStream outputStream = new ByteArrayOutputStream();  
 PrintStream originalOut = System.*out*; // Зберігаємо оригінальний потік виведення  
 System.*setOut*(new PrintStream(outputStream)); // Перенаправляємо потік виведення  
  
 // Виконання команди завантаження без встановленої кави  
 loadTruckCommand.execute();  
  
 // Перевірка, що виведено правильне повідомлення  
 String output = outputStream.toString().trim();  
 *assertEquals*("Необхідно встановити каву та кількість перед завантаженням.", output);  
  
 // Відновлення оригінального потоку виведення  
 System.*setOut*(originalOut);  
 }  
  
 @Test  
 void testExecute\_withoutQuantity() {  
 // Створення об'єкта кави, але без кількості  
 truck.model.Package package1 = new Package("Банка", 0.5, 1.0);  
 Coffee coffee = new Coffee("Robusta", 500, 100, 1, package1, 1);  
 loadTruckCommand.setCoffee(coffee, 0); // Кількість = 0  
  
 // Підготовка для тесту, коли кількість не була встановлена коректно  
 ByteArrayOutputStream outputStream = new ByteArrayOutputStream();  
 PrintStream originalOut = System.*out*; // Зберігаємо оригінальний потік виведення  
 System.*setOut*(new PrintStream(outputStream)); // Перенаправляємо потік виведення  
  
 // Виконання команди завантаження без встановленої кількості  
 loadTruckCommand.execute();  
  
 // Перевірка, що виведено правильне повідомлення  
 String output = outputStream.toString().trim();  
 *assertEquals*("Необхідно встановити каву та кількість перед завантаженням.", output);  
  
 // Відновлення оригінального потоку виведення  
 System.*setOut*(originalOut);  
 }  
}

package truck.command;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.\*;  
  
import truck.model.Truck;  
  
class SearchCoffeeCommandTest {  
  
 private Truck truck;  
 private SearchCoffeeCommand searchCoffeeCommand;  
  
 // Клас TruckMock для заміни реального Truck  
 static class TruckMock extends Truck {  
 boolean searchCoffeeCalled = false;  
  
 public TruckMock(double maxVolume) {  
 super(maxVolume);  
 }  
  
 @Override  
 public void searchCoffee() {  
 searchCoffeeCalled = true; // Поставимо прапор, коли метод буде викликаний  
 }  
 }  
  
 @BeforeEach  
 void setUp() {  
 // Створюємо заміну для Truck  
 truck = new TruckMock(10000);  
  
 // Створюємо команду  
 searchCoffeeCommand = new SearchCoffeeCommand(truck);  
 }  
  
 @Test  
 void testExecute\_searchCoffeeCalled() {  
 // Виконуємо команду  
 searchCoffeeCommand.execute();  
  
 // Перевіряємо, чи був викликаний метод searchCoffee  
 assertTrue(((TruckMock) truck).searchCoffeeCalled, "Метод searchCoffee не був викликаний");  
 }  
}

package truck.command;  
  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.\*;  
  
import truck.model.Shop;  
  
class ShowAvailableCoffeeCommandTest {  
  
 private Shop shop;  
 private ShowAvailableCoffeeCommand showAvailableCoffeeCommand;  
  
 // Створення заміни для класу Shop  
 static class ShopMock extends Shop {  
 boolean showAvailableCoffeeCalled = false;  
  
 @Override  
 public void showAvailableCoffee() {  
 showAvailableCoffeeCalled = true; // Прапор для перевірки, чи був викликаний метод  
 }  
 }  
  
 @BeforeEach  
 void setUp() {  
 // Створюємо заміну для Shop  
 shop = new ShopMock();  
  
 // Створюємо команду  
 showAvailableCoffeeCommand = new ShowAvailableCoffeeCommand(shop);  
 }  
  
 @Test  
 void testExecute\_showAvailableCoffeeCalled() {  
 // Виконуємо команду  
 showAvailableCoffeeCommand.execute();  
  
 // Перевіряємо, чи був викликаний метод showAvailableCoffee  
 *assertTrue*(((ShopMock) shop).showAvailableCoffeeCalled, "Метод showAvailableCoffee не був викликаний");  
 }  
}

package truck.command;  
  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import static org.junit.jupiter.api.Assertions.\*;  
  
import truck.model.Truck;  
  
class SortCoffeeCommandTest {  
  
 private Truck truck;  
 private SortCoffeeCommand sortCoffeeCommand;  
  
 // Створення заміни для класу Truck  
 static class TruckMock extends Truck {  
 boolean sortCoffeeCalled = false;  
  
 public TruckMock(double maxVolume) {  
 super(maxVolume);  
 }  
  
 @Override  
 public void sortCoffee() {  
 sortCoffeeCalled = true; // Прапор для перевірки, чи був викликаний метод  
 }  
 }  
  
 @BeforeEach  
 void setUp() {  
 // Створюємо заміну для Truck  
 truck = new TruckMock(10000);  
  
 // Створюємо команду  
 sortCoffeeCommand = new SortCoffeeCommand(truck);  
 }  
  
 @Test  
 void testExecute\_sortCoffeeCalled() {  
 // Виконуємо команду  
 sortCoffeeCommand.execute();  
  
 // Перевіряємо, чи був викликаний метод sortCoffee  
 *assertTrue*(((TruckMock) truck).sortCoffeeCalled, "Метод sortCoffee не був викликаний");  
 }  
}

package truck.model;  
  
import org.junit.jupiter.api.Test;  
import truck.command.Command;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
// Проста реалізація Command для тестування  
class LoadCommand implements Command {  
 private boolean executed = false;  
  
 @Override  
 public void execute() {  
 executed = true;  
 }  
  
 public boolean isExecuted() {  
 return executed;  
 }  
}  
  
class SearchCommand implements Command {  
 private boolean executed = false;  
  
 @Override  
 public void execute() {  
 executed = true;  
 }  
  
 public boolean isExecuted() {  
 return executed;  
 }  
}  
  
class SortCommand implements Command {  
 private boolean executed = false;  
  
 @Override  
 public void execute() {  
 executed = true;  
 }  
  
 public boolean isExecuted() {  
 return executed;  
 }  
}  
  
public class CarrierTest {  
  
 @Test  
 public void testLoadTruck() {  
 LoadCommand loadCommand = new LoadCommand();  
 SearchCommand searchCommand = new SearchCommand();  
 SortCommand sortCommand = new SortCommand();  
  
 Carrier carrier = new Carrier(loadCommand, searchCommand, sortCommand);  
  
 // Перевірка, що до виконання команди loadTruck вона ще не виконана  
 assertFalse(loadCommand.isExecuted());  
  
 // Виконання методу loadTruck  
 carrier.loadTruck();  
  
 // Перевірка, що команда була виконана  
 assertTrue(loadCommand.isExecuted());  
 }  
  
 @Test  
 public void testSearchCoffee() {  
 LoadCommand loadCommand = new LoadCommand();  
 SearchCommand searchCommand = new SearchCommand();  
 SortCommand sortCommand = new SortCommand();  
  
 Carrier carrier = new Carrier(loadCommand, searchCommand, sortCommand);  
  
 // Перевірка, що до виконання команди searchCoffee вона ще не виконана  
 assertFalse(searchCommand.isExecuted());  
  
 // Виконання методу searchCoffee  
 carrier.searchCoffee();  
  
 // Перевірка, що команда була виконана  
 assertTrue(searchCommand.isExecuted());  
 }  
  
 @Test  
 public void testSortCoffee() {  
 LoadCommand loadCommand = new LoadCommand();  
 SearchCommand searchCommand = new SearchCommand();  
 SortCommand sortCommand = new SortCommand();  
  
 Carrier carrier = new Carrier(loadCommand, searchCommand, sortCommand);  
  
 // Перевірка, що до виконання команди sortCoffee вона ще не виконана  
 assertFalse(sortCommand.isExecuted());  
  
 // Виконання методу sortCoffee  
 carrier.sortCoffee();  
  
 // Перевірка, що команда була виконана  
 assertTrue(sortCommand.isExecuted());  
 }  
  
 @Test  
 public void testCarrierBehavior() {  
 LoadCommand loadCommand = new LoadCommand();  
 SearchCommand searchCommand = new SearchCommand();  
 SortCommand sortCommand = new SortCommand();  
  
 Carrier carrier = new Carrier(loadCommand, searchCommand, sortCommand);  
  
 // Перевірка початкового стану  
 assertFalse(loadCommand.isExecuted());  
 assertFalse(searchCommand.isExecuted());  
 assertFalse(sortCommand.isExecuted());  
  
 // Виконання всіх команд  
 carrier.loadTruck();  
 carrier.searchCoffee();  
 carrier.sortCoffee();  
  
 // Перевірка, що всі команди були виконані  
 assertTrue(loadCommand.isExecuted());  
 assertTrue(searchCommand.isExecuted());  
 assertTrue(sortCommand.isExecuted());  
 }  
}

package truck.model;  
  
import org.junit.jupiter.api.Test;  
import truck.command.Command;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
// Проста реалізація Command для тестування  
class LoadCommand implements Command {  
 private boolean executed = false;  
  
 @Override  
 public void execute() {  
 executed = true;  
 }  
  
 public boolean isExecuted() {  
 return executed;  
 }  
}  
  
class SearchCommand implements Command {  
 private boolean executed = false;  
  
 @Override  
 public void execute() {  
 executed = true;  
 }  
  
 public boolean isExecuted() {  
 return executed;  
 }  
}  
  
class SortCommand implements Command {  
 private boolean executed = false;  
  
 @Override  
 public void execute() {  
 executed = true;  
 }  
  
 public boolean isExecuted() {  
 return executed;  
 }  
}  
  
public class CarrierTest {  
  
 @Test  
 public void testLoadTruck() {  
 LoadCommand loadCommand = new LoadCommand();  
 SearchCommand searchCommand = new SearchCommand();  
 SortCommand sortCommand = new SortCommand();  
  
 Carrier carrier = new Carrier(loadCommand, searchCommand, sortCommand);  
  
 // Перевірка, що до виконання команди loadTruck вона ще не виконана  
 *assertFalse*(loadCommand.isExecuted());  
  
 // Виконання методу loadTruck  
 carrier.loadTruck();  
  
 // Перевірка, що команда була виконана  
 *assertTrue*(loadCommand.isExecuted());  
 }  
  
 @Test  
 public void testSearchCoffee() {  
 LoadCommand loadCommand = new LoadCommand();  
 SearchCommand searchCommand = new SearchCommand();  
 SortCommand sortCommand = new SortCommand();  
  
 Carrier carrier = new Carrier(loadCommand, searchCommand, sortCommand);  
  
 // Перевірка, що до виконання команди searchCoffee вона ще не виконана  
 *assertFalse*(searchCommand.isExecuted());  
  
 // Виконання методу searchCoffee  
 carrier.searchCoffee();  
  
 // Перевірка, що команда була виконана  
 *assertTrue*(searchCommand.isExecuted());  
 }  
  
 @Test  
 public void testSortCoffee() {  
 LoadCommand loadCommand = new LoadCommand();  
 SearchCommand searchCommand = new SearchCommand();  
 SortCommand sortCommand = new SortCommand();  
  
 Carrier carrier = new Carrier(loadCommand, searchCommand, sortCommand);  
  
 // Перевірка, що до виконання команди sortCoffee вона ще не виконана  
 *assertFalse*(sortCommand.isExecuted());  
  
 // Виконання методу sortCoffee  
 carrier.sortCoffee();  
  
 // Перевірка, що команда була виконана  
 *assertTrue*(sortCommand.isExecuted());  
 }  
  
 @Test  
 public void testCarrierBehavior() {  
 LoadCommand loadCommand = new LoadCommand();  
 SearchCommand searchCommand = new SearchCommand();  
 SortCommand sortCommand = new SortCommand();  
  
 Carrier carrier = new Carrier(loadCommand, searchCommand, sortCommand);  
  
 // Перевірка початкового стану  
 *assertFalse*(loadCommand.isExecuted());  
 *assertFalse*(searchCommand.isExecuted());  
 *assertFalse*(sortCommand.isExecuted());  
  
 // Виконання всіх команд  
 carrier.loadTruck();  
 carrier.searchCoffee();  
 carrier.sortCoffee();  
  
 // Перевірка, що всі команди були виконані  
 *assertTrue*(loadCommand.isExecuted());  
 *assertTrue*(searchCommand.isExecuted());  
 *assertTrue*(sortCommand.isExecuted());  
 }  
}

package truck.model;  
  
import org.junit.jupiter.api.Test;  
import truck.command.Command;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
// Проста реалізація Command для тестування  
class LoadCommand implements Command {  
 private boolean executed = false;  
  
 @Override  
 public void execute() {  
 executed = true;  
 }  
  
 public boolean isExecuted() {  
 return executed;  
 }  
}  
  
class SearchCommand implements Command {  
 private boolean executed = false;  
  
 @Override  
 public void execute() {  
 executed = true;  
 }  
  
 public boolean isExecuted() {  
 return executed;  
 }  
}  
  
class SortCommand implements Command {  
 private boolean executed = false;  
  
 @Override  
 public void execute() {  
 executed = true;  
 }  
  
 public boolean isExecuted() {  
 return executed;  
 }  
}  
  
public class CarrierTest {  
  
 @Test  
 public void testLoadTruck() {  
 LoadCommand loadCommand = new LoadCommand();  
 SearchCommand searchCommand = new SearchCommand();  
 SortCommand sortCommand = new SortCommand();  
  
 Carrier carrier = new Carrier(loadCommand, searchCommand, sortCommand);  
  
 // Перевірка, що до виконання команди loadTruck вона ще не виконана  
 *assertFalse*(loadCommand.isExecuted());  
  
 // Виконання методу loadTruck  
 carrier.loadTruck();  
  
 // Перевірка, що команда була виконана  
 *assertTrue*(loadCommand.isExecuted());  
 }  
  
 @Test  
 public void testSearchCoffee() {  
 LoadCommand loadCommand = new LoadCommand();  
 SearchCommand searchCommand = new SearchCommand();  
 SortCommand sortCommand = new SortCommand();  
  
 Carrier carrier = new Carrier(loadCommand, searchCommand, sortCommand);  
  
 // Перевірка, що до виконання команди searchCoffee вона ще не виконана  
 *assertFalse*(searchCommand.isExecuted());  
  
 // Виконання методу searchCoffee  
 carrier.searchCoffee();  
  
 // Перевірка, що команда була виконана  
 *assertTrue*(searchCommand.isExecuted());  
 }  
  
 @Test  
 public void testSortCoffee() {  
 LoadCommand loadCommand = new LoadCommand();  
 SearchCommand searchCommand = new SearchCommand();  
 SortCommand sortCommand = new SortCommand();  
  
 Carrier carrier = new Carrier(loadCommand, searchCommand, sortCommand);  
  
 // Перевірка, що до виконання команди sortCoffee вона ще не виконана  
 *assertFalse*(sortCommand.isExecuted());  
  
 // Виконання методу sortCoffee  
 carrier.sortCoffee();  
  
 // Перевірка, що команда була виконана  
 *assertTrue*(sortCommand.isExecuted());  
 }  
  
 @Test  
 public void testCarrierBehavior() {  
 LoadCommand loadCommand = new LoadCommand();  
 SearchCommand searchCommand = new SearchCommand();  
 SortCommand sortCommand = new SortCommand();  
  
 Carrier carrier = new Carrier(loadCommand, searchCommand, sortCommand);  
  
 // Перевірка початкового стану  
 *assertFalse*(loadCommand.isExecuted());  
 *assertFalse*(searchCommand.isExecuted());  
 *assertFalse*(sortCommand.isExecuted());  
  
 // Виконання всіх команд  
 carrier.loadTruck();  
 carrier.searchCoffee();  
 carrier.sortCoffee();  
  
 // Перевірка, що всі команди були виконані  
 *assertTrue*(loadCommand.isExecuted());  
 *assertTrue*(searchCommand.isExecuted());  
 *assertTrue*(sortCommand.isExecuted());  
 }  
}

package truck.model;  
  
import org.junit.jupiter.api.Test;  
import truck.command.Command;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
// Проста реалізація Command для тестування  
class LoadCommand implements Command {  
 private boolean executed = false;  
  
 @Override  
 public void execute() {  
 executed = true;  
 }  
  
 public boolean isExecuted() {  
 return executed;  
 }  
}  
  
class SearchCommand implements Command {  
 private boolean executed = false;  
  
 @Override  
 public void execute() {  
 executed = true;  
 }  
  
 public boolean isExecuted() {  
 return executed;  
 }  
}  
  
class SortCommand implements Command {  
 private boolean executed = false;  
  
 @Override  
 public void execute() {  
 executed = true;  
 }  
  
 public boolean isExecuted() {  
 return executed;  
 }  
}  
  
public class CarrierTest {  
  
 @Test  
 public void testLoadTruck() {  
 LoadCommand loadCommand = new LoadCommand();  
 SearchCommand searchCommand = new SearchCommand();  
 SortCommand sortCommand = new SortCommand();  
  
 Carrier carrier = new Carrier(loadCommand, searchCommand, sortCommand);  
  
 // Перевірка, що до виконання команди loadTruck вона ще не виконана  
 *assertFalse*(loadCommand.isExecuted());  
  
 // Виконання методу loadTruck  
 carrier.loadTruck();  
  
 // Перевірка, що команда була виконана  
 *assertTrue*(loadCommand.isExecuted());  
 }  
  
 @Test  
 public void testSearchCoffee() {  
 LoadCommand loadCommand = new LoadCommand();  
 SearchCommand searchCommand = new SearchCommand();  
 SortCommand sortCommand = new SortCommand();  
  
 Carrier carrier = new Carrier(loadCommand, searchCommand, sortCommand);  
  
 // Перевірка, що до виконання команди searchCoffee вона ще не виконана  
 *assertFalse*(searchCommand.isExecuted());  
  
 // Виконання методу searchCoffee  
 carrier.searchCoffee();  
  
 // Перевірка, що команда була виконана  
 *assertTrue*(searchCommand.isExecuted());  
 }  
  
 @Test  
 public void testSortCoffee() {  
 LoadCommand loadCommand = new LoadCommand();  
 SearchCommand searchCommand = new SearchCommand();  
 SortCommand sortCommand = new SortCommand();  
  
 Carrier carrier = new Carrier(loadCommand, searchCommand, sortCommand);  
  
 // Перевірка, що до виконання команди sortCoffee вона ще не виконана  
 *assertFalse*(sortCommand.isExecuted());  
  
 // Виконання методу sortCoffee  
 carrier.sortCoffee();  
  
 // Перевірка, що команда була виконана  
 *assertTrue*(sortCommand.isExecuted());  
 }  
  
 @Test  
 public void testCarrierBehavior() {  
 LoadCommand loadCommand = new LoadCommand();  
 SearchCommand searchCommand = new SearchCommand();  
 SortCommand sortCommand = new SortCommand();  
  
 Carrier carrier = new Carrier(loadCommand, searchCommand, sortCommand);  
  
 // Перевірка початкового стану  
 *assertFalse*(loadCommand.isExecuted());  
 *assertFalse*(searchCommand.isExecuted());  
 *assertFalse*(sortCommand.isExecuted());  
  
 // Виконання всіх команд  
 carrier.loadTruck();  
 carrier.searchCoffee();  
 carrier.sortCoffee();  
  
 // Перевірка, що всі команди були виконані  
 *assertTrue*(loadCommand.isExecuted());  
 *assertTrue*(searchCommand.isExecuted());  
 *assertTrue*(sortCommand.isExecuted());  
 }  
}

package truck.model;  
  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
class PackageTest {  
  
 private Package package1;  
  
 @BeforeEach  
 void setUp() {  
 // Ініціалізуємо об'єкт Package перед кожним тестом  
 package1 = new Package("Пластик", 500, 1.2);  
 }  
  
 @Test  
 void testGetMaterial() {  
 // Перевіряємо, чи повертає гетер правильний матеріал  
 *assertEquals*("Пластик", package1.getMaterial(), "Матеріал упаковки має бути 'Пластик'");  
 }  
  
 @Test  
 void testGetWeight() {  
 // Перевіряємо, чи повертає гетер правильну вагу  
 *assertEquals*(500, package1.getWeight(), "Вага упаковки має бути 500 грамів");  
 }  
  
 @Test  
 void testGetVolume() {  
 // Перевіряємо, чи повертає гетер правильний об'єм  
 *assertEquals*(1.2, package1.getVolume(), "Об'єм упаковки має бути 1.2 літра");  
 }  
  
 @Test  
 void testToString() {  
 // Перевіряємо, чи правильно працює метод toString  
 String expected = "Упаковка: матеріал = Пластик, вага = 500,00 г, об'єм = 1,20 л";  
 *assertEquals*(expected, package1.toString(), "Метод toString має повертати правильний рядок");  
 }  
}

package truck.model;  
  
import org.junit.jupiter.api.Test;  
import truck.command.Command;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
// Проста реалізація Command для тестування  
class AddCommand implements Command {  
 private boolean executed = false;  
  
 @Override  
 public void execute() {  
 executed = true;  
 }  
  
 public boolean isExecuted() {  
 return executed;  
 }  
}  
  
class ShowCommand implements Command {  
 private boolean executed = false;  
  
 @Override  
 public void execute() {  
 executed = true;  
 }  
  
 public boolean isExecuted() {  
 return executed;  
 }  
}  
  
public class SellerTest {  
  
 @Test  
 public void testAddCoffee() {  
 AddCommand addCommand = new AddCommand();  
 ShowCommand showCommand = new ShowCommand();  
  
 Seller seller = new Seller(showCommand, addCommand);  
  
 // Перевірка, що до виконання команди addCoffee вона ще не виконана  
 *assertFalse*(addCommand.isExecuted());  
  
 // Виконання методу addCoffee  
 seller.addCoffee();  
  
 // Перевірка, що команда була виконана  
 *assertTrue*(addCommand.isExecuted());  
 }  
  
 @Test  
 public void testShowCoffee() {  
 AddCommand addCommand = new AddCommand();  
 ShowCommand showCommand = new ShowCommand();  
  
 Seller seller = new Seller(showCommand, addCommand);  
  
 // Перевірка, що до виконання команди showCoffee вона ще не виконана  
 *assertFalse*(showCommand.isExecuted());  
  
 // Виконання методу showCoffee  
 seller.showCoffee();  
  
 // Перевірка, що команда була виконана  
 *assertTrue*(showCommand.isExecuted());  
 }  
  
 @Test  
 public void testSellerBehavior() {  
 AddCommand addCommand = new AddCommand();  
 ShowCommand showCommand = new ShowCommand();  
  
 Seller seller = new Seller(showCommand, addCommand);  
  
 // Перевірка початкового стану  
 *assertFalse*(addCommand.isExecuted());  
 *assertFalse*(showCommand.isExecuted());  
  
 // Виконання всіх команд  
 seller.addCoffee();  
 seller.showCoffee();  
  
 // Перевірка, що всі команди були виконані  
 *assertTrue*(addCommand.isExecuted());  
 *assertTrue*(showCommand.isExecuted());  
 }  
}

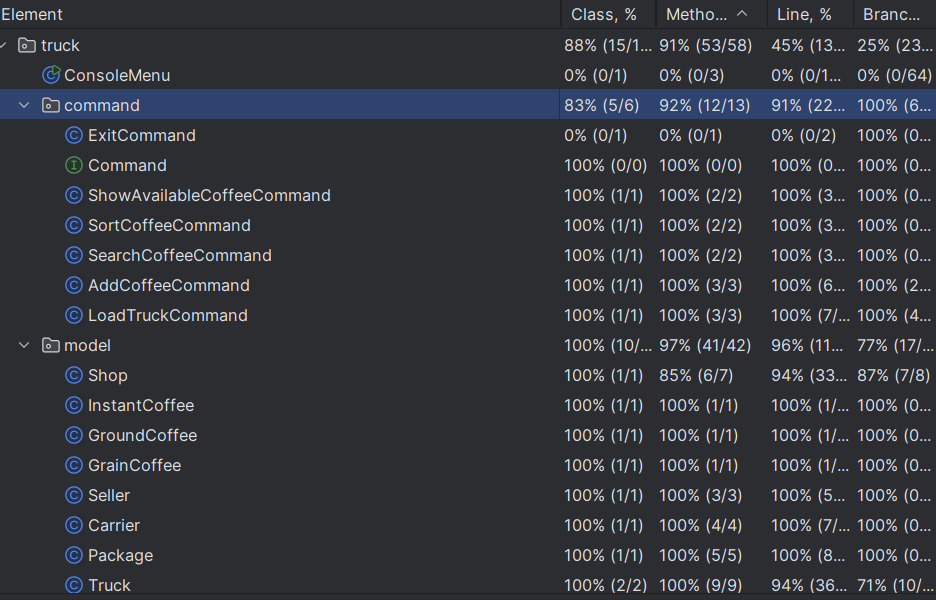
package truck.model;  
  
import org.junit.jupiter.api.Test;  
import truck.command.Command;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
// Проста реалізація Command для тестування  
class AddCommand implements Command {  
 private boolean executed = false;  
  
 @Override  
 public void execute() {  
 executed = true;  
 }  
  
 public boolean isExecuted() {  
 return executed;  
 }  
}  
  
class ShowCommand implements Command {  
 private boolean executed = false;  
  
 @Override  
 public void execute() {  
 executed = true;  
 }  
  
 public boolean isExecuted() {  
 return executed;  
 }  
}  
  
public class SellerTest {  
  
 @Test  
 public void testAddCoffee() {  
 AddCommand addCommand = new AddCommand();  
 ShowCommand showCommand = new ShowCommand();  
  
 Seller seller = new Seller(showCommand, addCommand);  
  
 // Перевірка, що до виконання команди addCoffee вона ще не виконана  
 *assertFalse*(addCommand.isExecuted());  
  
 // Виконання методу addCoffee  
 seller.addCoffee();  
  
 // Перевірка, що команда була виконана  
 *assertTrue*(addCommand.isExecuted());  
 }  
  
 @Test  
 public void testShowCoffee() {  
 AddCommand addCommand = new AddCommand();  
 ShowCommand showCommand = new ShowCommand();  
  
 Seller seller = new Seller(showCommand, addCommand);  
  
 // Перевірка, що до виконання команди showCoffee вона ще не виконана  
 *assertFalse*(showCommand.isExecuted());  
  
 // Виконання методу showCoffee  
 seller.showCoffee();  
  
 // Перевірка, що команда була виконана  
 *assertTrue*(showCommand.isExecuted());  
 }  
  
 @Test  
 public void testSellerBehavior() {  
 AddCommand addCommand = new AddCommand();  
 ShowCommand showCommand = new ShowCommand();  
  
 Seller seller = new Seller(showCommand, addCommand);  
  
 // Перевірка початкового стану  
 *assertFalse*(addCommand.isExecuted());  
 *assertFalse*(showCommand.isExecuted());  
  
 // Виконання всіх команд  
 seller.addCoffee();  
 seller.showCoffee();  
  
 // Перевірка, що всі команди були виконані  
 *assertTrue*(addCommand.isExecuted());  
 *assertTrue*(showCommand.isExecuted());  
 }  
}

package truck.model;  
  
import org.junit.jupiter.api.Test;  
import truck.command.Command;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
// Проста реалізація Command для тестування  
class AddCommand implements Command {  
 private boolean executed = false;  
  
 @Override  
 public void execute() {  
 executed = true;  
 }  
  
 public boolean isExecuted() {  
 return executed;  
 }  
}  
  
class ShowCommand implements Command {  
 private boolean executed = false;  
  
 @Override  
 public void execute() {  
 executed = true;  
 }  
  
 public boolean isExecuted() {  
 return executed;  
 }  
}  
  
public class SellerTest {  
  
 @Test  
 public void testAddCoffee() {  
 AddCommand addCommand = new AddCommand();  
 ShowCommand showCommand = new ShowCommand();  
  
 Seller seller = new Seller(showCommand, addCommand);  
  
 // Перевірка, що до виконання команди addCoffee вона ще не виконана  
 *assertFalse*(addCommand.isExecuted());  
  
 // Виконання методу addCoffee  
 seller.addCoffee();  
  
 // Перевірка, що команда була виконана  
 *assertTrue*(addCommand.isExecuted());  
 }  
  
 @Test  
 public void testShowCoffee() {  
 AddCommand addCommand = new AddCommand();  
 ShowCommand showCommand = new ShowCommand();  
  
 Seller seller = new Seller(showCommand, addCommand);  
  
 // Перевірка, що до виконання команди showCoffee вона ще не виконана  
 *assertFalse*(showCommand.isExecuted());  
  
 // Виконання методу showCoffee  
 seller.showCoffee();  
  
 // Перевірка, що команда була виконана  
 *assertTrue*(showCommand.isExecuted());  
 }  
  
 @Test  
 public void testSellerBehavior() {  
 AddCommand addCommand = new AddCommand();  
 ShowCommand showCommand = new ShowCommand();  
  
 Seller seller = new Seller(showCommand, addCommand);  
  
 // Перевірка початкового стану  
 *assertFalse*(addCommand.isExecuted());  
 *assertFalse*(showCommand.isExecuted());  
  
 // Виконання всіх команд  
 seller.addCoffee();  
 seller.showCoffee();  
  
 // Перевірка, що всі команди були виконані  
 *assertTrue*(addCommand.isExecuted());  
 *assertTrue*(showCommand.isExecuted());  
 }  
}

package truck.model;  
  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
import truck.model.Coffee;  
import truck.model.Shop;  
import truck.model.GrainCoffee;  
import truck.model.Package;  
  
import java.io.FileWriter;  
import java.io.IOException;  
import java.util.List;  
import static org.junit.jupiter.api.Assertions.\*;  
  
class ShopTest {  
 private Shop shop;  
  
 @BeforeEach  
 void setUp() {  
 shop = new Shop();  
 }  
  
 @Test  
 void testInitializeDefaultCoffees() {  
 // Перевіряємо, що за замовчуванням додається 3 види кави  
 List<Coffee> coffees = shop.getAvailableCoffee();  
 *assertEquals*(3, coffees.size(), "Кількість кави за замовчуванням має бути 3");  
 }  
  
 @Test  
 void testAddCoffee() {  
 Package package1 = new Package("Банка", 0.5, 1.0);  
 Coffee newCoffee = new GrainCoffee("Нова зернова кава", 500, 120, 1.2, package1, 1000);  
  
 shop.addCoffee(newCoffee);  
 List<Coffee> coffees = shop.getAvailableCoffee();  
  
 *assertTrue*(coffees.contains(newCoffee), "Кава повинна бути додана до списку доступної кави");  
 *assertEquals*(4, coffees.size(), "Кількість кави після додавання має бути 4");  
 }  
  
 @Test  
 void testShowAvailableCoffeeWhenNotEmpty() {  
 // Захоплюємо стандартний вивід  
 List<Coffee> coffees = shop.getAvailableCoffee();  
 *assertFalse*(coffees.isEmpty(), "Список кави не має бути порожнім");  
  
 // Викликаємо метод showAvailableCoffee і переконуємося, що він відображає доступну каву  
 shop.showAvailableCoffee();  
 }  
  
 @Test  
 void testShowAvailableCoffeeWhenEmpty() {  
 // Очищаємо список кави, щоб перевірити випадок з порожнім списком  
 shop.getAvailableCoffee().clear();  
  
 *assertTrue*(shop.getAvailableCoffee().isEmpty(), "Список кави має бути порожнім");  
  
 // Викликаємо метод showAvailableCoffee і переконуємося, що вивід правильний  
 shop.showAvailableCoffee();  
 }  
  
 @Test  
 void testLoadCoffeesFromFile() throws IOException {  
 // Створюємо тестовий файл для завантаження  
 String testFilePath = "test\_coffee\_file.txt";  
 try (FileWriter writer = new FileWriter(testFilePath)) {  
 writer.write("Зернова кава,500,100,1,Банка,0.5,1.0,1000\n");  
 writer.write("Мелена кава,260,50,0.5,Пакет,0.2,0.5,500\n");  
 writer.write("Розчинна кава,100,30,0.25,Пакет,0.2,0.5,2000\n");  
 }  
  
 // Перевіряємо початкову кількість кави  
 int initialSize = shop.getAvailableCoffee().size();  
  
 // Завантажуємо каву з файлу  
 shop.loadCoffeesFromFile(testFilePath);  
  
 // Перевіряємо, що кількість кави збільшилася  
 *assertEquals*(initialSize + 3, shop.getAvailableCoffee().size(), "Кількість кави має збільшитися на 3 після завантаження з файлу");  
  
 // Видаляємо тестовий файл після тесту  
 new java.io.File(testFilePath).delete();  
 }  
}

package truck.model;  
  
import org.junit.jupiter.api.BeforeEach;  
import org.junit.jupiter.api.Test;  
  
import java.io.ByteArrayInputStream;  
import java.io.InputStream;  
import java.util.List;  
  
import static org.junit.jupiter.api.Assertions.\*;  
  
class TruckTest {  
 private Truck truck;  
 private Coffee coffee1;  
 private Coffee coffee2;  
 private Coffee coffee3;  
  
 @BeforeEach  
 void setUp() {  
 // Ініціалізуємо фургон з максимальною ємністю 500 літрів  
 truck = new Truck(500);  
  
 // Ініціалізуємо зразки кави  
 coffee1 = new Coffee("Arabica", 1.0, 100.0, 2.0, new Package("Plastic", 0.1, 2.0), 10);  
 coffee2 = new Coffee("Robusta", 0.5, 50.0, 1.0, new Package("Metal", 0.05, 1.0), 20);  
 coffee3 = new Coffee("Liberica", 2.0, 200.0, 3.0, new Package("Glass", 0.2, 3.0), 5);  
 }  
  
 @Test  
 void testLoadTruck\_Success() {  
 // Завантажуємо фургон  
 truck.loadTruck(coffee1, 5);  
  
 // Перевіряємо, що кількість кави у фургоні оновилася  
 *assertEquals*(5, coffee1.getQuantityInTruck());  
  
 // Перевіряємо, що об'єм фургона оновився  
 *assertTrue*(truck.getCurrentVolume() <= truck.getMaxVolume());  
 }  
  
 @Test  
 void testLoadTruck\_NotEnoughSpace() {  
 // Спробуємо завантажити більше, ніж максимальний об'єм фургона  
 truck.loadTruck(coffee3, 20000);  
  
 // Перевіряємо, що кількість кави у фургоні не змінилася  
 *assertEquals*(0, coffee3.getQuantityInTruck());  
  
 // Перевіряємо, що об'єм фургона не перевищив максимальний об'єм  
 *assertTrue*(truck.getCurrentVolume() <= truck.getMaxVolume());  
 }  
  
 @Test  
 void testSortCoffee() {  
 // Додаємо декілька видів кави у фургон  
 truck.loadTruck(coffee1, 5);  
 truck.loadTruck(coffee2, 10);  
 truck.loadTruck(coffee3, 2);  
  
 // Виконуємо сортування кави  
 truck.sortCoffee();  
  
 // Перевіряємо порядок сортування за співвідношенням ціни до ваги  
 List<Coffee> loadedCoffees = truck.getLoadedCoffees();  
 *assertTrue*(loadedCoffees.get(0).getPrice() / loadedCoffees.get(0).getWeight()  
 <= loadedCoffees.get(1).getPrice() / loadedCoffees.get(1).getWeight());  
 }  
  
 @Test  
 void testSearchCoffee\_Found() {  
 // Додаємо каву у фургон  
 truck.loadTruck(coffee1, 5);  
  
 // Встановлюємо введення для емуляції пошуку кави  
 String input = "Arabica\n";  
 InputStream in = new ByteArrayInputStream(input.getBytes());  
 System.*setIn*(in);  
  
 // Перевіряємо пошук кави  
 truck.searchCoffee();  
  
 // Перевіряємо, що кава знайдена  
 *assertTrue*(truck.getLoadedCoffees().stream().anyMatch(coffee -> coffee.getName().equals("Arabica")));  
 }  
  
 @Test  
 void testSearchCoffee\_NotFound() {  
 // Встановлюємо введення для емуляції пошуку кави  
 String input = "NonExistingCoffee\n";  
 InputStream in = new ByteArrayInputStream(input.getBytes());  
 System.*setIn*(in);  
  
 // Перевіряємо, що кава не знайдена  
 truck.searchCoffee();  
  
 // Перевіряємо, що кава не була додана  
 *assertFalse*(truck.getLoadedCoffees().stream().anyMatch(coffee -> coffee.getName().equals("NonExistingCoffee")));  
 }  
  
 @Test  
 void testGetCurrentVolume() {  
 // Завантажуємо фургон, щоб збільшити об'єм  
 truck.loadTruck(coffee1, 5);  
  
 // Перевіряємо, що getCurrentVolume повертає коректний об'єм  
 double expectedVolume = coffee1.getTotalVolume() \* 5;  
 *assertEquals*(expectedVolume, truck.getCurrentVolume(), 0.001);  
 }  
  
 @Test  
 void testGetLoadedCoffees() {  
 // Завантажуємо фургон  
 truck.loadTruck(coffee1, 5);  
  
 // Перевіряємо, що завантажена кава міститься в списку  
 List<Coffee> loadedCoffees = truck.getLoadedCoffees();  
 *assertTrue*(loadedCoffees.contains(coffee1));  
 }  
  
 @Test  
 void testGetMaxVolume() {  
 // Перевіряємо, що getMaxVolume повертає правильний максимальний об'єм  
 *assertEquals*(500, truck.getMaxVolume(), 0.001);  
 }  
}

**Аналіз покриття**



**Висновок**

В ході цієї лабораторної роботи я навчився робити юніт тести для усіх класів програми. Було покрито усі методи моєї програми на 90%.

Посилання на Git: https://github.com/Float36/labs